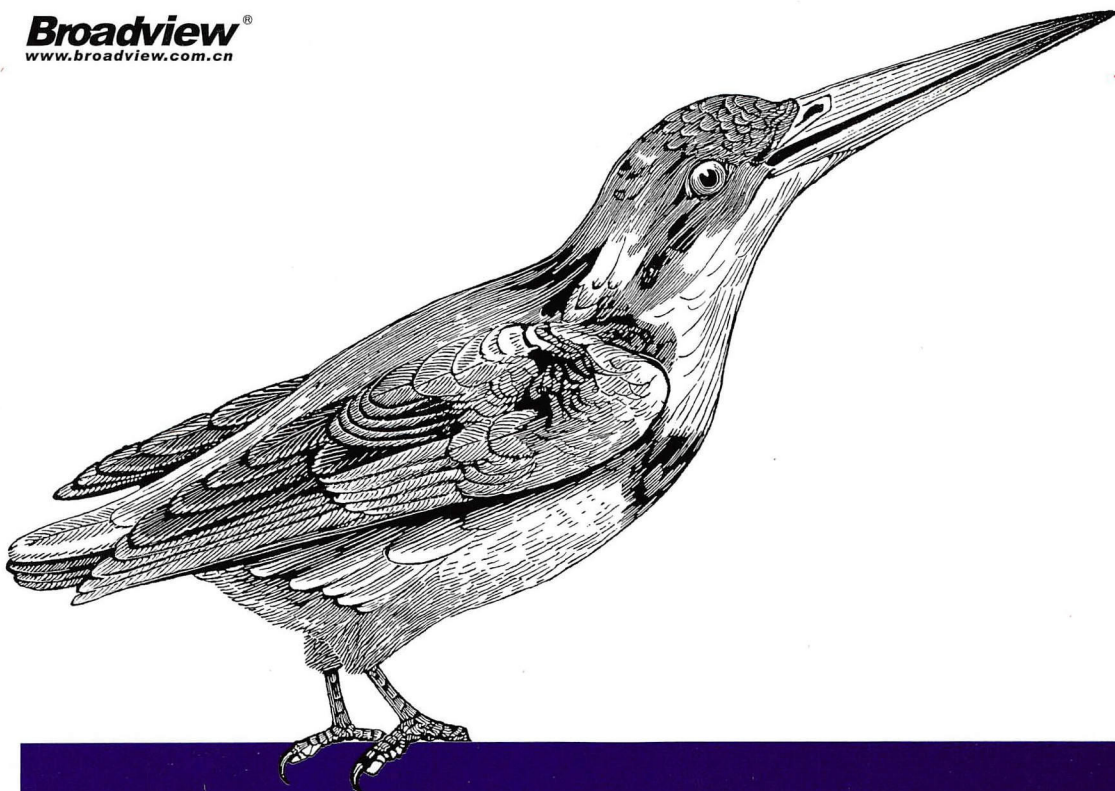


版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF

O'REILLY®

Broadview®
www.broadview.com.cn



云原生Java

Spring Boot、Spring Cloud与Cloud Foundry弹性系统设计

Cloud Native Java

[美] Josh Long Kenny Bastani 著
张若飞 宋净超 译



 中国工信出版集团

 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

O'REILLY®

云原生Java

Spring Boot、Spring Cloud与Cloud Foundry
弹性系统设计

Cloud Native Java

[美] Josh Long Kenny Bastani 著

张若飞 宋净超 译

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING



内 容 简 介

无论是传统 IT 行业，还是互联网行业，都正处于行业历史上最剧烈的变革中：大量的系统正在从传统的 IT 架构转向基于云的架构，开发模式也正在从开发和运维分工的传统模式，逐渐转向统一的 DevOps 模式。Java 技术也应运进入了新的生命周期，大量被用于构建现代的、基于云的应用程序。

本书深入研究了云计算、测试驱动开发、微服务与持续集成和持续交付领域的工具和方法，并指导你将传统应用程序转变为真正的云原生应用程序。其中重点介绍了微服务框架 Spring Boot，以及如何使用 Spring Boot 轻松创建任何粒度的 Spring 服务，并部署到现代的容器环境中。可以说本书是一本讲述如何使用 Spring Boot、Spring Cloud 和 Cloud Foundry 构建软件的理论 and 实践的完备指南。

本书主要面向正在使用 Spring Boot、Spring Cloud 和 Cloud Foundry 构建软件的 Java/JVM 开发人员。

©2017 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2018. Authorized translation of the English edition, 2017 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

本书简体中文版专有出版权由 O'Reilly Media, Inc. 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2017-8746

图书在版编目 (CIP) 数据

云原生 Java: Spring Boot、Spring Cloud 与 Cloud Foundry 弹性系统设计 / (美) 乔西·朗 (Josh Long), (美) 肯尼·巴斯塔尼 (Kenny Bastani) 著; 张若飞, 宋净超译. —北京: 电子工业出版社, 2018.6

书名原文: Cloud Native Java

ISBN 978-7-121-34251-6

I. ①云… II. ①乔… ②肯… ③张… ④宋… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字 (2018) 第 106114 号

策划编辑: 张春雨

责任编辑: 牛 勇

封面设计: Karen Montgomery 张 健

印 刷: 三河市良远印务有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787 × 980 1/16

印张: 36.25

字数: 794 千字

版 次: 2018 年 6 月第 1 版

印 次: 2018 年 6 月第 1 次印刷

定 价: 128.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888, 88258888。

质量投诉请发邮件至 zlt@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。



O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal



译者序

“Java 已死。”

这几年，我们不断听到这句话，甚至相信过这句话，认为 Java 语言已经存在了如此长的时间，影响了如此广的范围，是时候该有一些新鲜的东西来颠覆它了。但是令人失望的是，Java 语言不仅自身依然在快速地进化（本书完成时，Java 10 已经正式发布），其整个生态圈也都不曾停下前进的脚步。Spring 就是一个很好的例子。相信 Java 开发人员对它已经再熟悉不过了，大部分人每天的工作都是在围绕着它进行开发。令人惊讶的是，Spring 也已经从以前单一的核心框架，逐渐发展成为一个全栈的应用层框架，甚至在大家还没有觉察的时候，它已经开始用于构建微服务的生态系统了，而 Spring Boot 就是整个微服务生态的核心。

在 Spring Boot 出现之前，我们看到 Spring 也在努力尝试，将整个繁杂的 Spring/Spring MVC 体系精简成一个容易上手的工具，也出现了像 Spring Roo 这样的过渡产品。但是直到 Spring Boot 横空出世，我们才发现，原来开发一个应用程序，可以如此简单。这使我不由想起 10 年前，我翻译了国内第一本 Grails 书籍和 Rails 等书籍，当时那些令我陶醉的 feature，这一次终于踏踏实实地在 Spring Boot 中得到了实现。与此同时，Spring 团队秉承了其一贯的风格，简单却强大。如今，Spring Boot 不再是一个简单的框架，而是像一个灵活的骨架结构，你可以很轻松地将 Spring Data、Spring Security 这些耳熟能详的模块，像插积木一样插在 Spring Boot 上，甚至不需要写一行代码。它就像一幅绝世的中国山水画，让你既惊叹于其精致的笔法，又被它整体磅礴的气势震慑得不能一语。而这一切，都与 Java 这十年间不断的提升和整个 Spring 生态的发展密不可分。

如今的 IT 行业，已经不再是十年前的情景，云计算已经发展成熟并且逐渐占据主流。没有人愿意再去花高额的成本、时间、人手建立自己的数据中心，基于云计算的产品和项目甚至一天就可以开发完成上线。更快，永远是技术追逐的目标。所有你能想到的框架、组件、中间件，甚至是业务功能模块，都在快速地被云端化，你随便打开一个云计算产品的官网，上面都有不下上百种的产品功能供你选择，即便是最新的机器学习平台，你



也只需点几下鼠标就可以搭建起来。更简单，也永远是技术发展的趋势。在 Netflix 的成功背景下，Spring 没有停留在 Spring Boot 上止步不前，而是进一步迈向了云原生的应用开发，因此而诞生了 Spring Cloud。从严格意义上说，Spring Cloud 才是一个真正的微服务框架，它提供了一套完善的解决方案，能够将云环境中各个独立部署的微服务整合起来，因此云原生的应用，本质上就是一个分布式的应用。同时，它打破了我们之前依靠硬件或其他一些软件来实现分布式系统的做法，将其一切功能都云端化，包括中心化配置、负载均衡、路由、集群，以及衍生出来的断路器等常见设计模式。我们恍然发现，原来开发一个云上的应用也变得如此简单，但其功能依然强大。可以预见的是，掌握 Spring Boot、Spring Cloud 的开发技术，以及核心思想，一定会是下一代 Java 开发人员必备的技能。幸运的是，我们已经在当前的工作中实际运用了它们，并取得了非常不错的效果。

感谢 Josh 和 Kenny 分享了他们各自在这个领域的深刻理解和经验，编纂成此书，同时我也为能够参与翻译此书感到荣幸。相信这本书，可以为 Java 开发人员打开下一个时代的大门，让大家在原生云环境的应用中徜徉。感谢张春雨编辑的耐心和督促，不知不觉合作已十年有余。感谢净超（Jimmy）临危受命，承担了本书后半部分的翻译工作，他的高产和高效令我刮目相看，也时刻能感受到他对技术饱含的热情。感谢我身边的所有朋友，一直在支持着我，鼓励着我，这也是当我面对压力和挫折时不放弃的理由。最后，感谢我的妻子，给了我莫大的支持和鼓励，让我可以投身创业之中，愿她一生幸福满足！感恩我的母亲，让我懂得了生命如此脆弱又如此坚强，生死之外，再无大事，愿她一生健康平安！

译者 张若飞

2018 年 5 月 23 日凌晨 于北京



致 Eva 和 Makani，爱你们的 Josh 叔叔。

致我的祖父 Abbas，他等了一百年，终于等到在一本书的封面上看到自己的名字。

——Kenny



目录

序言 (James Watters)	xvii
序言 (Rod Johnson)	xix
前言	xxi

第 I 部分 基础知识

第 1 章 云原生应用程序	3
亚马逊的故事	3
平台的承诺	5
模式	7
可扩展性	7
可靠性	8
敏捷性	8
Netflix 的故事	9
微服务	11
拆分单体系统	12
Netflix OSS	13
云原生 Java	14
十二要素原则	14
代码库	15
依赖	15
配置	16



后端服务	17
构建、发布、运行	17
进程	17
端口绑定	18
并发	18
易处理	18
开发 / 生产环境一致	19
日志	19
管理进程	19
总结	20
第 2 章 训练营：Spring Boot 和 Cloud Foundry	21
什么是 Spring Boot	21
Spring Initializr 入门	21
Spring Tool Suite 入门	30
安装 Spring Tool Suite (STS)	30
使用 Spring Initializr 创建一个新项目	31
Spring 指南大全	36
遵循 STS 中的指南	38
配置	40
Cloud Foundry 平台	52
总结	66
第 3 章 符合十二要素程序风格的配置	67
令人迷惑的“配置”合并	67
Spring 框架对配置的支持	67
PropertyPlaceholderConfigurer	68
Environment 接口和 @Value 注解	69
Profile	71
启动配置	73
使用 Spring Cloud Config Server 进行中心化、日志型的配置	76
Spring Cloud Config Server	76
Spring Cloud Config 客户端	78
安全	79

可刷新的配置	79
总结	83
第 4 章 测试	85
测试的构成	86
在 Spring Boot 中进行测试	86
集成测试	88
测试切片	89
测试中的 Mock	89
使用 @SpringBootTest 中的 Servlet 容器	93
测试分片	94
端到端测试	102
测试分布式系统	102
消费者驱动的契约测试	104
Spring Cloud Contract	105
总结	113
第 5 章 迁移遗留的应用程序	115
契约	115
迁移应用程序环境	116
开箱即用的构建包 (Buildpacks)	116
自定义的构建包	117
容器化的应用程序	118
将应用程序迁移到云上的微重构	119
连接后端服务	120
用 Spring 实现服务平等	121
总结	133

第 II 部分 Web 服务

第 6 章 REST API	137
伦纳德·理查森的成熟模型	137
使用 Spring MVC 实现简单的 REST API	139
内容协商	142

读写二进制数据	142
Google Protocol Buffers	145
错误处理	150
超媒体	152
媒体类型和模式	158
API 版本	159
编写 REST API 文档	162
客户端	167
用于临时浏览和交互的 REST 客户端	167
RestTemplate	171
总结	177
第 7 章 路由	179
DiscoveryClient 接口	180
Cloud Foundry Route 服务	190
总结	195
第 8 章 边缘服务	197
Greetings 服务	198
一个简单的边缘服务	200
Netflix Feign	202
使用 Netflix Zuul 进行过滤和代理	204
自定义 Zuul 过滤器	214
边缘服务的安全	218
OAuth	219
服务端应用程序	220
HTML5 和 JavaScript 单页面应用程序	221
没有用户的应用	221
受信任的客户端	221
Spring Security	222
Spring Cloud Security	227
一个 Spring Security OAuth 授权服务器	227
保护 Greetings 资源服务器的安全	232
创建一个受 OAuth 保护的单页面应用程序	238
总结	247

第Ⅲ部分 数据整合

第9章 数据管理	251
数据建模	251
关系数据库管理系统 (RDBMS)	252
NoSQL	253
Spring Data	253
Spring Data 应用程序的结构	254
域类	254
库	254
为领域数据组织 Java 包	255
使用 JDBC 访问 RDBMS 数据	258
Spring 的 JDBC 支持	259
Spring Data 示例	261
Spring Data JPA	264
Account Service	264
集成测试	274
Spring Data MongoDB	275
Order Service	275
集成测试	282
Spring Data Neo4j	284
Inventory Service	284
集成测试	294
Spring Data Redis	297
高速缓存	298
总结	302
第10章 消息系统	303
Spring Integration 的事件驱动架构	304
消息端点	305
使用简单的组件构建复杂的系统	306
消息代理、桥接、竞争消费者模式和事件溯源	314
发布—订阅目的地	314
点对点目的地	315

Spring Cloud Stream	315
流生产者	316
流消费者	321
总结	323
第 11 章 批处理和任务	325
批处理工作	325
Spring Batch.....	326
我们的第一个批处理作业	327
调度	336
通过消息传递远程分区 Spring 批处理作业	337
任务管理	346
通过 Workflow 进行的以工作流为中心的整合	348
使用消息传递的分布式	362
总结	362
第 12 章 数据集成	363
分布式事务	364
故障隔离和优雅的降级	364
saga 模式	369
CQRS (命令查询责任分离)	369
投诉 API	371
投诉统计 API	383
Spring Cloud Data Flow	385
Stream	387
任务	390
REST API	391
实现 Data Flow 客户端	392
总结	407

第 IV 部分 生产

第 13 章 可观测的系统	411
你构建，你运行	412

谋杀神秘微服务	413
十二要素运维	413
新方式	414
可观测性	416
推与拉的可观测性和解析率	416
使用 Spring Boot Actuator 捕获应用程序的当前状态	417
度量	418
通过 /info 端点识别服务	431
健康检查	432
审计事件	436
应用程序日志	439
指定日志输出	440
指定日志级别	441
分布式跟踪	445
用 Spring Cloud Sleuth 寻找线索	446
多少数据是足够的	447
OpenZipkin：一张图片胜过千丝万缕	448
跟踪其他平台和技术	454
仪表板	455
使用 Hystrix 仪表板监控下游服务	455
Codecentric 的 Spring Boot Admin	459
Ordina Microservices 仪表板	462
Pivotal Cloud Foundry 的 AppsManager	463
修复	465
总结	467
第 14 章 服务代理	469
创建后台服务	470
平台视图	472
使用 Spring Cloud Cloud Foundry Service Broker 实现服务代理	473
简单的 Amazon S3 服务代理	473
服务目录	474
管理服务实例	476
服务绑定	482
保护服务代理	486

部署	487
使用 BOSH 发布	487
使用 Cloud Foundry 发布	488
注册 Amazon S3 Service Broker	489
创建 Amazon S3 服务实例	490
消费服务实例	491
S3 客户端应用程序	493
运行测试	496
总结	496
第 15 章 持续交付	497
持续集成之外	497
John Allspaw 在 Flickr 以及后来的 Etsy	498
Netflix 的 Adrian Cockroft	499
亚马逊的持续交付	500
流水线	500
测试	501
持续交付微服务	502
工具	503
Concourse	503
容器	504
持续交付微服务	504
安装 Concourse	505
基本的管道设计	506
持续集成	518
消费者驱动的协约测试	518
User 微服务流水线	519
数据	522
生产	523

第 V 部分 附录

附录 A 在 Java EE 中使用 Spring Boot	527
---	------------

序言 (James Watters)

你不能两次踏入同一条河流。

——赫拉克利特 (希腊哲学家)

2015 年夏天，在威尼斯海滩的一家咖啡店里，我坐在 Josh Long 旁边，我知道我们要开始干一件大事了。因为开发者不断要求了解我们的新技术——Spring Boot，所以他的行程几乎都被排满了。我们的云原生平台 Pivotal Cloud Foundry，正在迅速成为流行的云原生应用程序运行时环境。随着 Spring Boot 越来越受欢迎，Spring Cloud 的到来有望成为一个爆炸点。“这将是一件大事，而且正在发生。”我跟他说。

工作中的力量是巨大的。在 CIO 全力寻求如何提升开发者生产力的时候，Spring Boot 提供了一个微服务和 DevOps 友好的企业级开发方式。随着 Spring Boot 和 PCF 之间的集成日益完善，生产环境部署成为一个简单的流水线和 API 调用工作。Spring Cloud 不仅提供了世界上第一个微服务网格，并且诞生了一种新的、基于云的 Java 标准。

这些技术的独特组合，不仅仅是表面看上去开发风格的变化，而是改变了大型组织的交付结构。由于对传统 Java 应用程序来说，服务器和运维的复杂性太高，因此开发人员被禁止与生产环境接触。我们经常听到，客户部署一次程序要花费几天这样恐怖的故事。我们知道我们的平台会改变他们的生活。有一些粉丝客户开始给我们写邮件，描述他们在 PCF 上采用 Spring Boot 之后，如何将生产环境的更新从几个月的时间，降低到几分钟之内。

自 2015 年以来，现实已经证明，所有迁移到这种开发方式的企业，开发速度至少提高了 50%，超过了 MTTR (平均维护时间) / 停机时间的一半，并且他们与许多小型的平台团队一起运行了成千上万的 JVM 实例。最重要的是，采用云原生 Java 的企业，可以拿出更多的时间来思考他们的客户和市场，同时对开发和运维复杂性的担忧大大减少。

本书对现代企业软件设计中重要的模式，逐一进行了详细的论述。在许多例子中，你可

以看到 Josh 和 Kenny 与许多世界顶级企业一同努力所带来的实践经验。

我建议每个开发者和 IT 主管，做好充分的准备，挖掘企业的适应性和柔韧性，来享受这份工作。

*James Watters, Pivotal Cloud Foundry 高级副总裁，
Pivotal, @wattersjames*

序言 (Rod Johnson)

我们正处于行业历史上最剧烈的变革中：从传统架构转向云的架构，从传统开发和运维分工转向统一的 DevOps。本书可帮助你进行转型，在本书中详细阐述了开发云原生应用程序的机遇和挑战，并明确指出了成功实现的方向。

转型不会一夜之间发生。这本书最好的一点，是它强调了如何将以现有经验构建的环境，逐步迁移到云上。特别是第 5 章中“用 Spring 实现服务平等”一节，提供了一个传统企业转向云端的优秀实战案例。

本书是理论和实践的完美结合，既解释了现代应用的架构原理，又给出了有效的、经过验证的实现方法。实践需要我们做出选择，不只是编程语言的选择，更主要是选择开源框架，因为现代应用程序几乎都是建立在通用问题的开源解决方案之上的。如果你选择了 Java 语言，或是对编程语言持开放态度——本书就是你需要的书。

我的死亡报告是夸大的。

——马克·吐温

几年前，报道 Java 死亡的消息不绝于耳。今天，Java 仍在蓬勃发展，本书会告诉你为什么。Java 已经进入了新的生命周期，部分原因是因为 Java 技术已经用于构建现代的、基于云的应用程序了。使 Netflix 成功的两个关键因素，就是开源项目和 Spring。本书在这两个方面都做得非常出色。

Spring 最初的核心思想，是降低过去 Java EE 的复杂性，它经受住了时间的考验，成为了云应用的完美基础。十多年前，我们曾经谈论过“Spring 三原则”：依赖注入、可移植服务抽象和 AOP。如今，将业务逻辑与环境完美分离比以往任何时候都重要，所有这一切都没有发生变化。

本书的重点是介绍 Spring Boot，它是在微服务时代一种使用 Spring 的新方式，我们无法拒绝使用它。Spring Boot 可以轻松创建任何粒度的 Spring 服务，并部署到现代的容

器环境中。传统的“企业级”Java 应用程序，都是只能运行在更大服务器上的单体程序，而 Spring Boot 以它的简单和效率改变了这一切：服务聚焦精准，以足够的服务器来运行。

书中的案例展示了 Spring 团队的最新工作，例如，Spring Cloud Stream 和改进的集成测试支持，以及与 Netflix 开源项目不断的集成。

我很高兴看到 Spring 的持续创新，以及专注于简单化开发人员的工作。虽然在过去的 5 年中，我只是以一个用户的身份与 Spring 进行了互动，但是看到它繁荣发展，成功解决了许多复杂的问题，还是很高兴的。今天，我仍然在 Atomist 从事这类工作，我们的目标是让所有事情自动化，让 Spring 来为所有的开发团队和开发过程做 Java 应用程序的事情。Spring 为 Java 开发人员所关心的每件事情，都提供了一个简单的、有效的结构和抽象，同时，Atomist 也希望为项目源代码、系统构建、问题跟踪器、环境部署等提供同样的东西。无论是为了提高团队协作能力而使用 Slack 管理项目，还是为了监控部署事件，Atomist 都可以提供强大的开发自动化能力。

自动化的基本组成部分是测试。我特别喜欢本书对测试的介绍，书中对如何解决微服务测试中的许多难题提出了宝贵的意见。我也喜欢书中许多注释详尽的代码清单，作者很好地坚持了这个 O'Reilly 一贯的风格。

我非常荣幸，可以为同道好友作序。Josh 是一个善于沟通的人。他的文字功底和他现场编写的代码一样好。Josh 和 Kenny 是两位充满激情的、好奇并且见多识广的导游，很高兴与他们一起走过这个旅程。我在旅途中学到了很多，我相信你也会学到很多。

*Rod Johnson, Spring Framework、Atomist 创始人兼 CEO,
twitter 账号 @springrod*

前言

更快! 更快! 更快!! 每个人都想走得更快, 但很少人知道如何做到。市场的需求在不断增长, 机会也越来越多, 但我们中的一些人根本无法跟上节奏! 传统企业与亚马逊, 以及 Netflix 和 Etsy 之间的区别是什么? 我们知道这些公司已经成长到拥有令人疯狂的规模, 但不知什么原因, 它们仍然能够保持竞争优势, 保持领先地位。它们究竟是如何做到的?

一个想法从概念到实现, 需要做大量的工作。要了解一个想法, 既要看它的效用, 也要看它的价值。这些工作要经历很多不同的环节, 从涉及用户体验的产品管理到测试, 再到运行, 最终才进入生产环境。从历史的角度看, 这其中的每一道工序都会让整个工作变慢。我们作为一个开源社区, 随着时间的推移, 已经优化了这个流程中的一些部分。我们有了云计算, 所以不再需要机架和机柜。我们使用测试驱动开发和持续集成来实现自动化测试。我们以小步迭代的方式来发布软件, 通过微服务来减小代码变更的范围和成本。我们拥抱 DevOps 背后的理念(武装的同理心)来增强对整个系统的了解, 增进开发人员与运维人员之间的感情, 减少不同优先级事项之间的巨大协同成本。这些事情本身并不有趣, 改进也不大, 但是如果将它们结合起来, 我们就可以将整个价值链中的每件重要事情都隔离开来。总而言之, 这些东西就是我们所说的云原生。

作为行业中的一员和理论的实践者, 软件开发人员今天处于一个激情的时代。我们在基础设施、测试、中间件、持续集成和交付、开发框架和云平台等方面, 都拥有可靠的、开源的、稳定和自助服务的解决方案。这些基础条件让企业可以专注于如何低成本地提供高商业价值, 以及扩张到更大的规模。

谁应该阅读本书

本书主要面向正在使用 Spring Boot、Spring Cloud 和 Cloud Foundry, 以更快、更好地构建软件的 Java / JVM 开发人员。相信你已经听说过微服务的概念, 也许你已经看到了

Spring Boot 的不断发展，并且想知道为什么今天大多数企业都在使用 Cloud Foundry。那么本书可以告诉你答案。

为什么我们写这本书

在 Pivotal，我们通过传授持续交付的知识，以及通过 Cloud Foundry、Spring Boot 和 Spring Cloud 来帮助客户向数字化企业转型。我们已经知道了什么是可行的（以及什么不可行），并且希望把客户的实践和我们的经验总结出来。我们并不想面面俱到，但我们试图清晰地向你介绍整个云原生世界的关键概念。

浏览本书

本书的组织结构如下：

- 第 1 章和第 2 章介绍了云原生思想产生的背景，然后介绍了 Spring Boot 和 Cloud Foundry
- 第 3 章介绍了如何配置 Spring Boot 应用程序。这是我们所要依赖的基本技能。
- 第 4 章介绍了如何测试 Spring 应用程序，从如何测试最简单的组件到测试分布式系统。
- 第 5 章介绍了可以将应用程序迁移到 Cloud Foundry 等云平台的轻量级重构方式，你可以从中获得一些有价值的额外经验。
- 第 6 章介绍了如何使用 Spring 构建 HTTP 和 RESTful 服务。你会在 API 和领域驱动开发中用到其中很多技巧。
- 第 7 章介绍了在分布式系统中控制请求进出的常用方法。
- 第 8 章介绍了如何构建一个响应外部请求的服务。
- 第 9 章介绍了如何使用 Spring Data 在 Spring 中管理数据。这为领域驱动的思想奠定了基础。
- 第 10 章介绍了如何使用 Spring 中事件驱动、消息中心化的能力，来集成分布式服务和数据。
- 第 11 章介绍了如何利用云平台（如 Cloud Foundry）的能力来处理长期运行的工作。
- 第 12 章介绍了在分布式系统中管理状态的一些方法。
- 第 13 章介绍了如何构建具备可观测性和可操作性的系统。
- 第 14 章介绍了如何构建类似于 Cloud Foundry 平台的服务代理。服务代理可以将有状态的服务（例如消息队列、数据库和缓存）连接到云平台。
- 第 15 章介绍了持续交付背后的思想。这虽然是最后一章，但也可能是你旅程的开始。

如果你像我们一样，你不会从前到后来阅读本书。如果你真的像我们一样，你通常不会阅读前言。但是，既然你看到了这里，我们给出以下一些建议：

- 无论你是做什么的，请阅读第 1 章和第 2 章。这两章为本书的其余部分奠定了基础。如果没有第 1 章中所介绍的动机和业务背景，本书中所有的技术讨论就都没有了意义。而所有的技术讨论都依赖于在第 2 章中所建立的基础。
- 第 3 ~ 6 章介绍了任何 Spring 开发人员都应该注意的事项。这些概念不仅适用于较旧版本的 Spring 应用程序，也适用于新的应用程序。第 5 章介绍了如何同时兼容新旧版本的应用程序（无论是否使用了 Spring）。
- 第 7 章和第 8 章介绍了一些基于 HTTP 的微服务系统中的概念，包括安全性和路由。
- 第 9 ~ 12 章可以帮助你更好地管理和处理分布式系统中的数据。
- 第 13 章介绍了一个真正的核心概念，因为它依赖于其他一些技术概念，所以我们在本书前半部分介绍核心概念和测试时没有介绍它。可运维的应用程序应该是可观测的。尽早了解本章的基本原理，会帮助你理解本书的其他内容。
- 第 14 章介绍了如何使用 Spring（一个云原生的开发框架）来构建平台和云端的组件。本章对开放服务代理的讨论尤为深刻。
- 最后，第 15 章提炼了有关持续交付的知识。整本书是按照持续交付的方式编写的，这对于我们正在努力做的事情至关重要，因为我们选择用结果来证明一切。务必阅读本章。

在线资源

我们提供了很多有用的在线资源来帮助你理解书中的内容：

- 本书的代码可以在 GitHub 资料库（<http://github.com/cloud-native-java/>）中找到。
- 从 Spring 网站（<http://spring.io/>）上能找到关于 Spring 的一切资料，包括文档、技术问答论坛等。
- Cloud Foundry 网站（<http://cloudfoundry.org>）是由 Cloud Foundry 基金会的所有贡献者通力完成的。你会在上面找到相关的视频、教程、新闻等。

本书使用约定

本书使用以下印刷约定。

斜体 (*Italic*)

斜体表示新的术语、URL、电子邮件地址、文件名和文件扩展名等。

等宽字体 (Constant width)

用于程序清单, 以及段落中引用的程序元素, 例如变量或函数名、数据库、数据类型、环境变量、语句和关键字等。

等宽字体加粗 (Constant width bold)

显示应该由用户输入的命令或其他文本。

等宽斜体 (Constant width italic)

该处内容应该被由用户提供的值或者上下文所确定的值替换。



表示一个提示或建议。



表示一般注释。



表示警告或注意。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室 (100035)
奥莱利技术咨询 (北京) 有限公司

O'Reilly 的每一本书都有专属网站，你可以在那里找到关于本书的相关信息，包括勘误列表、示例代码以及其他信息。本书的网站地址是：

<http://bit.ly/cloud-native-java>

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于我们的书籍、课程、会议和新闻的更多信息，请参阅我们的网站 <http://www.oreilly.com>。

在 Facebook 上找到我们：<http://facebook.com/oreilly>

在 Twitter 上关注我们：<http://twitter.com/oreillymedia>

在 YouTube 上观看我们：<http://www.youtube.com/oreillymedia>

使用代码示例

你可以从 <http://github.com/Cloud-Native-Java> 下载补充材料（代码示例、练习等）。

这本书是为了帮助你完成工作而编写的。一般来说，如果本书提供了示例代码，那么你可以在程序和文档中使用它们。除非你大量使用本书中的代码，否则不需要与我们联系。例如，编写一段使用了本书中多段代码的程序不需要我们授权。销售或者发行 O'Reilly 书中代码示例的 CD-ROM 需要授权。引用本书内容和示例代码来回答问题不需要授权。将本书中重要的示例代码用于产品文档中需要授权。

使用我们的代码时，希望你能标明它的出处。出处一般要包含书名、作者、出版商和 ISBN，例如，“Book Title by Josh Long and Kenny Bastani (O'Reilly). Copyright 2017 Josh Long, Kenny Bastani, 978-1-449-37464-8.”。

如果还有其他使用代码的情形需要与我们沟通，可以随时与我们联系：permissions@oreilly.com。

致谢

首先，我们要感谢 O'Reilly 出版社的编辑 Nan Barber 和 Brian Foster，他们付出了令人难以置信的耐心，并给了我们很大的支持。

感谢所有给我们提供观点、想法和灵感的审稿人员。非常感谢 Pivotal 公司和它的庞大商业生态给我们的支持。特别感谢提供技术反馈的每位审稿人员，其中包括 Brian

Dus-sault、Dave Syer 博士、Andrew Clay Shafer、Rob Winch、Mark Fisher 博士、Mark Pollack 博士、Utkarsh Nadkarni、Sabby Anandan、Michael Hunger、Bridget Kromhout、Russ Miles、Mark Heckler、Marten Deinum、Nathaniel Schutta 和 Patrick Crocker，等等。谢谢！

最后，我们要感谢 Rod Johnson 和 James Watters。你们不求回报，给予了我们一切需要的帮助。非常感谢你们的序言、反馈和灵感。

本书是使用 AsciiDoctor 工具编写的，该工具由 OpenDevise 的 Dan Allen (<http://twitter.com/mojavelinux>) 和设计合作伙伴 Sarah White (<http://twitter.com/carbonfray>) 领导开发。所有示例的源代码存放在 GitHub (<http://github.com>) 的公共仓库中。代码持续集成使用了 Travis CI (<https://travis-ci.org/cloud-native-java>)。构建产出物托管在由 JFrog (<https://www.jfrog.com>) 慷慨捐赠给我们的 Artifactory 仓库中。所有的代码示例都可以在由 Pivotal 托管的 Pivotal Web Services 上运行。如果没有这些工具，编写本书不会如此顺利，无论是开源社区还是由社区运维的公司，都在免费支持我们。非常感谢你们！我们希望你的下一个项目会考虑使用他们的工具，并支持他们，因为他们曾经支持过我们。

想对 Spring 团队(<http://spring.io/team>)和 Cloud Foundry 团队(<http://cloud.foundry.org>)说，非常感谢你们为我们编写所有代码、进行测试和对我们予以所有支持。

Josh Long

我要感谢我的合著者 Kenny，加入我的这次冒险旅程！我想感谢 O'Reilly 给我们这次机会，以及他们给了我们难以置信的宽容，因为我们为了能够完整介绍 Pivotal 的生态系统而一再延期。谢谢 Dave Syer 博士 (https://twitter.com/david_syer) 对本书致言和给予我鼓励。不管在任何城市、时区、国家和大洲，我都在不断编程，而 Pivotal 的 Spring 和 Cloud Foundry 团队无论何时都在帮助我，谢谢你们！

Kenny Bastani

首先，我要感谢我的朋友、导师和同事 Michael Hunger，他首先激发了我为开源软件写书和发声的热情。我也要感谢我非常才华的合著者 Josh Long，当我两年前把我的第一个 Spring Boot 微服务投入生产时，他邀请我和他一起写这本书。与他一起写这本书是一次令人难以置信的冒险。从 Josh 和我在纸上写下第一行字起，我们看到了 Spring Boot 逐渐成长为最成功的开源项目之一。今天，这种增长意味着每月 Spring Boot 被下载 1300 万次，其中包括新的应用程序和持续的生产环境部署。达到这些成绩，Spring

团队已经发展到有将近 50 名全职工程师。保守估计，Spring 团队维护着 100 多个开源项目，其中包括框架组件、示例应用程序和文档。所有这些项目都有赞助商 Pivotal 的支持。如果没有这些工作人员的巨大奉献，我们无法完成本书，因为他们所支持的开发者社区，每年产生约 300 万个新的、可运行于生产环境的 Spring Boot 应用程序。

从 2004 年到 2017 年，仅仅 Spring 框架项目，已经收到 381 个开源 Java 开发者的 36412 个提交，这大概相当于 339 年的人力成本。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- 提交勘误：你对书中内容的修改意见可在[提交勘误处](#)提交，若被采纳，将获赠博文视点社区积分（在购买电子书时，积分可用来抵扣相应金额）。
- 交流互动：在页面下方[读者评论处](#)留下你的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34251>



第 I 部分

基础知识

云原生应用程序

无论对于团队和个人，软件的开发模式总是在发展。开源软件运动为软件行业带来了工具、框架、平台和操作系统的爆炸式增长，所有这些都越来越注重灵活性和自动化。当今大多数比较受欢迎的开源工具都将功能集中在，如何让软件团队能够从开发到运维的各个层面上，不断提高持续交付软件的速度。

亚马逊的故事

从 20 世纪 90 年代初开始，经过 20 年的发展，总部设在西雅图的亚马逊已经从网络书店成长为世界上最大的在线零售商。今天，它就像著名的“亚马逊河”一样，销售的东西远不仅仅是书籍了。在 2015 年，亚马逊超过了沃尔玛成为美国最有价值的零售商。在亚马逊耀眼的成长故事中，最有趣的一部分可以简单地归纳成一个问题：作为一个简单的在线书店网站，在没有设立任何零售网点的前提下，是如何成为世界上最大的零售商之一的？

不难看出，商业世界在全球各地不断增长的互联网推动力下，围绕着数字连接进行着转变和重塑。随着个人电脑变得越来越小，变成了今天人们随时随地使用的智能手机、平板电脑和智能手表，我们经历了销售渠道的指数级增长，这正在改变世界的商业方式。

亚马逊的首席技术官 Werner Vogels，见证了亚马逊从一个非常成功的在线书店，到世界上最有价值公司和零售商之一的技术演进。2006 年 6 月，Vogels 在计算机杂志 *ACM Queue* 上接受了一段关于技术选型演进如何驱动亚马逊快速增长的采访。在采访中，Vogels 谈及其背后的核心驱动力。

亚马逊技术演进的很大一部分原因是为了实现持续的增长，在实现超大规模的同时保持可用性和性能。

——Werner Vogels, “A Conversation with Werner Vogels,” *ACM Queue* 4, no. 4 (2006):14–22.

Vogels 说，亚马逊为了实现超大规模，必须向不同的软件架构模式迁移。他提到亚马逊开始是一个单体应用程序。随着时间的推移，越来越多的团队在同一应用程序中运行代码，代码库的所有权界限也开始变得模糊。“没有隔离的结果，就是没有清晰的所有权。” Vogels 说道。

Vogels 指出，共享资源（例如数据库）使得整体业务难以扩展。无论是应用程序服务器还是数据库，共享资源的数量越多，团队对功能上线时的控制就越少。

你建立它，你运行它。

——Werner Vogels，亚马逊首席技术官

Vogels 谈及了云原生应用程序的本质：团队对所构建的代码全权负责。他继续说：“在传统的模式下，开发和运维之间是隔离的，你把软件往他们中间一扔就不管了。但是在亚马逊，你建立它，你运行它。”

在曾经世界首屈一指的软件大会上，由主要发言人发表的最重要的引用之一就是“你建立它，你运行它”。这个词已经成为我们今天所熟知的一个 *DevOps* 流行语。

Vogels 在 2006 年谈到的许多做法，已经成为了如今蓬勃发展的流行软件运动的种子。*DevOps* 和微服务等实践活动可以与 Vogels 在十年前提出的想法相互印证。虽然像亚马逊这样的大型互联网公司正在开发类似的工具，但这些工具需要许多年的开发才能成熟到对外提供服务。

亚马逊 2006 年推出了名为 Amazon Web Services (AWS) 的新产品。AWS 背后的想法是提供一个平台，同亚马逊内部使用的平台一样，并将其作为服务向公众发布。亚马逊希望将这个平台背后的想法和工具商品化。Vogels 提出的许多想法已经被应用到 Amazon.com 平台中。通过将平台作为服务向公众发布，亚马逊进入了一个称为公有云的新市场。

公有云背后的想法是很好的。用户可以按需配置虚拟资源，而无须担心基础架构。开发人员可以简单地租用虚拟机来运行他们的应用程序，而无须购买或管理基础设施。这是一种低风险的自助服务形式，有助于增加公有云的吸引力，也让 AWS 在行业中领先一步。

AWS 花了几年时间，才让构建公有云上服务和模式的方法变得成熟。虽然许多开发人员

通过这些服务来构建新的应用程序，但是许多现存公司仍然最关心迁移的问题。之前的应用程序不是为了便携性而设计的。此外，许多应用程序仍然依赖于与公有云不兼容的遗留功能。

为了让大多数大公司能够使用公有云，他们需要改变开发应用程序的方式。

平台的承诺

平台今天是一个被滥用的词。

当我们谈论计算机领域的平台时，我们一般指的是一组帮助我们构建或运行应用程序的功能。对平台的最好描述，就是它对开发人员构建应用程序提出了哪些约束。

平台能够自动执行应用程序核心业务需求之外的任务。这使得开发团队可以更加灵活，能够支持具有差异化商业价值的功能。

任何通过编写 shell 脚本、容器或虚拟机来进行自动部署的团队，都已经建立了一个平台。问题是，该平台保守的承诺是什么？如果要持续发布新软件的大多数（甚至所有）功能，它需要做多少工作？

当我们创建平台时，我们其实正在创建一个自动执行一组可重复实践行为的工具。这些实践行为，来自于我们将有价值的想法变成现实的过程中，所经历的一系列约束条件。

- **想法：**我们平台的核心思想是什么，为什么它们是有价值的？
- **约束条件：**将我们的想法转化为现实所必须面对的约束条件是什么？
- **实践行为：**我们如何将这些约束条件自动变成一组可重复的行为？

当我们认识到这一点后，每个平台的核心，就是通过自动化工具，来增加那些能够让商业价值差异化的简单想法。

我们以 Amazon.com 平台为例。Werner Vogels 表示，通过增加软件组件之间的隔离性，团队可以对发布到生产环境中的功能拥有更多的控制力。

想法：

- 通过增加软件组件之间的隔离性，我们能够快速、独立地交付系统的各个部分。

当我们把这个想法作为平台的基础后，可以为其设计一系列的约束条件。约束条件，就是当核心理念被实际自动化后，对如何创造价值的一个看法。以下是我们对如何增加组件隔离性提出的约束条件。

约束条件：

- 软件组件需要构建为可独立部署的服务。
- 服务中的所有业务逻辑都需要使用其运行的数据进行封装。
- 从服务外部无法直接访问数据库。
- 服务需要发布一个允许其他服务访问自身业务逻辑的 Web 界面。

有了这些约束条件之后，我们就对在实践中如何增加软件组件的隔离性问题，提出了自己的一个观点。当实际执行自动化时，由于有这些约束，从而为团队交付生产环境功能提供更多的控制力。下一步是描述如何将这组约束条件变成一组可重复的行为。

从这些约束条件中产生的实践行为，应该被描述为一组承诺的集合。通过将实践行为作为一种承诺，平台用户就会对如何构建和运维其应用程序有一个一致的认知。

实践行为：

- 为团队提供一个自助服务的界面，用于配置应用程序运维的基础架构。
- 通过自助服务界面，可以将应用程序打包，并发布到某一个环境上。
- 以服务的形式向应用程序提供数据库，并且可以使用自助服务界面进行配置。
- 以环境变量的形式，为应用程序提供一个数据库密码，但是只有明确声明它和数据库之间的关系后才可以作为服务绑定。
- 为每个应用程序都提供一个服务注册中心，用于查找外部所依赖的服务位置。

上面列出的每种行为都是对用户的一个承诺。通过这种方式，平台的核心想法就会变成现实中对应用程序的约束条件。

云原生应用程序就是建立在一系列的约束之上，从而减少了花费在同类型重复工作的时间。

当 AWS 首次向公众发布时，亚马逊并没有强制用户遵守他们在 Amazon.com 内部使用的相同约束条件。确实如 *Amazon Web Services* 的名称一般，AWS 本身并不是一个云平台，而更像是一个包含许多独立基础服务的集合，这些服务可以组合成一个用来配置一组承诺的自动化工具。在 AWS 首次发布的多年后，亚马逊开始提供一系列托管的平台服务，范围从物联网（IoT）到机器学习。

如果每个公司都需要从零开始构建自己的平台，那么就必须等待平台完成，从而延迟交付应用程序的时间。早期采用 AWS 的公司需要把很多自动化工具整合起来，才能形成一个平台。每个公司都不得不为如何开发和发布软件的想法，制定一套自己的承诺。

但是现在，软件行业的看法已经趋于一致，每个云平台都应该有一套基本的共同承诺。



本书将使用开源的 PaaS（平台即服务）平台 Cloud Foundry 来介绍这些承诺。Cloud Foundry 背后的核心目的是提供一个平台，能够为快速构建和运维系统提供一套常见的承诺。Cloud Foundry 在实现这些承诺的同时，还提供了在不同的云基础设施提供商之间的可移植性。

本书的主要内容是如何构建云原生的 Java 应用程序。我们重点关注一些工具和框架，它们能够通过利用云原生平台的承诺和优势，来帮助减少大量的重复工作。

模式

我们开发软件的新模式，使我们能够更多地思考应用程序如何在生产环境中运行。开发人员和运维人员一起协同，能够让彼此更好地理解应用程序在生产环境中的运行情况，提高发生故障时应对问题的能力。

与 Amazon.com 的情况一样，软件架构师也开始抛弃大型的单体应用程序。架构师们现在专注于如何实现超大规模下的可扩展性，同时又不牺牲性能和可用性。通过将单体应用程序拆分成独立的组件，工程化的企业正在努力将管理去中心化，从而为团队提供更多开发功能上的控制力。通过增加组件之间的隔离性，软件团队开始进入了分布式系统开发的阶段，聚焦如何在单个发布周期中去构建更小的、功能更单一的服务。

云原生的应用程序利用了这种模式，使团队能够更加敏捷地将功能发布到生产环境。随着应用程序变得更加分布式（为了给负责程序的团队提供更多的控制权，就必须增加系统之间的隔离性），应用程序组件之间通信失败的概率就变成了一个需要重视的问题。随着软件程序变成复杂的分布式系统，执行失败成为一个必然的结果。

云原生应用程序的架构在提供了超大规模可扩展性的同时，依然保证了应用程序整体的可用性和性能。虽然像亚马逊这样的公司在云上获得了超大规模可扩展性的优势，但还没有出现可用于构建云原生应用程序的大众工具。这类工具和平台最终会以公有云先驱 Netflix 所维护的一套开源项目集合展示到大众面前。

可扩展性

为了更快地开发软件，我们需要考虑软件在各个层次上的规模。从最普遍的意义上看，规模是一个计算产生价值的成本的函数，而会导致价值降低的不可预测的程度被称为风险。在这种情况下，我们不得不考虑规模，因为构建软件充满了风险。运维人员并不总是知道软件开发人员所创造的风险。当我们要求开发人员更快地将功能发布到生产环境时，我们正在增加运维人员的风险，但从来没有考虑运维人员的处境。



这样做的结果是，运维人员对开发者编写的软件产生不信任。开发人员和运维人员之间缺乏信任，互相责备。人们互相指责对方，而不是去思考问题产生的原因，或者至少想一些临时办法降低对业务的影响。

为了减轻对 IT 组织传统结构的压力，我们需要重新思考如何交付软件以及如何和运维人员沟通。运维和开发人员之间的沟通可能会影响我们系统的扩展能力，因为各方的目标会随着时间的推移而变得不一致。要想取得成功，需要采用更加可靠的软件开发方法——一方面要强调软件开发流程中运维团队的经验，一方面要加强团队之间的学习交流和能力提升。

可靠性

团队之间所产生的期望（开发、运维、用户体验等）是一种合约。团队之间创建的合约意味着互相提供或消费了某种程度上的服务。通过观察团队在开发软件过程中如何互相提供服务，我们可以更好地了解，服务间的通信失败如何会导致系统性整体故障。

为了降低意外风险对整体性商业价值造成的损失，我们需要在团队之间建立服务协议。明确团队之间的服务协议，是为了确保实际与预期的运维成本一致。这样，我们通过服务可以使企业单元的产出最大化。软件本身的商业目的就是通过成本可靠地预测价值产生量——我们称之为可靠性。

一个企业的服务模型和我们在构建软件时使用的模型是一样的。这也是为什么在我们的软件中，无论是自动化还是由人工手动执行，都能够保证系统的可靠性的原因。

敏捷性

我们开始发现，开发和运维软件已经不再只有一种方式。在敏捷方法和软件即服务(SaaS)模式的推动下，企业应用程序架构逐渐变成分布式的。开发分布式系统是一项复杂的任务。而对于采用分布式架构的公司来说，这种转变是为了更快地交付软件，以及降低故障风险。



可以听到你在说，“敏捷？不是敏捷已死吗？”(<https://www.linkedin.com/pulse/agile-dead-matthew-kern>)”敏捷，既指一个组织对交付新功能的热爱，也指要快速进行响应。我们这里只谈论对敏捷的看法，而不是从管理实践角度来讨论。交付产品有很多种方式，我们也不在乎你使用的是哪种管理方式。关键是，我们要知道敏捷是一个价值观，而不是目的。

如今大部分开发软件的企业，都在重新定义它的开发流程，以便能够更快地交付软件并



将应用程序持续部署到生产环境中。公司不仅希望提高软件的开发速度，还希望增加创建和运维应用程序的数量，以便更好地为企业各个业务部门提供服务。

软件已经逐渐成为企业的竞争优势。更好的工具能够让业务专家开辟新的收入来源，或者通过快速创新的方式来优化业务能力。

这个过程的核心就是云。当我们谈论云的时候，我们正在谈论一些非常具体的技术，其能够让开发人员和运维人员利用现有的 Web 服务来配置和管理虚拟化的计算型基础架构。

许多公司正在从数据中心迁移到公有云。其中一家公司就是著名的订阅式流媒体公司 Netflix。

Netflix 的故事

今天，Netflix 是世界上最大的点播流媒体服务之一，在云上运行着其所有的在线服务。1997 年，Netflix 由 Reed Hastings 和 Marc Randolph 成立于加利福尼亚州的斯科茨谷。最初 Netflix 提供了一个在线 DVD 租赁服务，允许客户每月支付固定的订阅费用，享受无限制的电影租赁服务，没有滞纳金。客户在 Netflix 网站上选择光盘并购买后，会收到用邮件发送的 DVD。

在 2008 年，Netflix 遭遇了严重的数据库损坏，导致公司无法向其客户发送任何 DVD。当时，Netflix 刚刚开始向客户提供流媒体视频服务。Netflix 的流媒体团队意识到，类似的流媒体故障对其业务的未来将是毁灭性的。因此，Netflix 做出了一个至关重要的决定：它将采用一种不同的方式来开发和运维软件，以确保其服务始终可以提供给客户使用。

在 Netflix 决定的避免在线服务故障的方案中，其中一点是它必须要抛弃只能垂直扩展的基础设施，以及避免单点故障。之所以认识到这一点，是因为发生故障的数据库就是一个只能垂直扩展的关系型数据库。Netflix 将客户数据迁移到一个分布式的 NoSQL 数据库，即一个名为 Apache Cassandra 的开源数据库项目。在迈出这一步后，Netflix 逐渐成为一个“云原生”的公司，将所有的软件应用程序运行在高度分布和具有弹性的云服务上。Netflix 采用了水平扩展的基础架构模式，通过为其应用和数据库不断增加冗余，来提高其在线服务的健壮性。

作为 Netflix 迁移到云服务决定的一部分，需要将现有的大型应用程序迁移到高度可靠的分布式系统。所面临的一个重大挑战是：Netflix 的团队必须重新构建他们的应用程序，同时将内部的数据中心迁移到公有云上。2009 年，Netflix 开始转向 Amazon Web



Services (AWS)，并聚焦三个主要目标：可扩展性、性能和可用性。

从 2009 年初开始，这种需求量急剧增加。事实上，Netflix 云平台工程部的副总裁 Yury Izrailevsky 在 2013 年的 AWS re:Invent 会议上提到，自 2009 年以来，Netflix 在 AWS 上的规模已经增加了 100 倍。“我们自己的解决方案无法扩展到如此的规模。”Izrailevsky 说道。

此外，Izrailevsky 还指出，在全球化快速扩张的今天，云服务良好的可扩展性所带来的优势更加明显。“为了给我们的欧洲客户提供更好的低延迟体验，我们在爱尔兰推出了第二个云服务区域。在不同国家搭建一个新的数据中心将需要几个月的时间和数百万美元。这将是一个巨大的投资。”他说。

随着 Netflix 开始在 Amazon Web Services 上托管其应用程序，员工们也开始在 Netflix 的公司博客上记录他们的学习过程。Netflix 的许多员工都主张采用一种新型的架构，来重点解决软件栈各层面的水平可扩展性。

Netflix 的个性化团队技术副总裁 John Ciancutti 在 2010 年底的公司博客中表示：“云计算环境是水平可扩展架构的理想选择。我们不必在几个月前就开始猜测我们的硬件、存储和网络需求将会如何增长。我们可以通过编程的方式，几乎立刻从 Amazon Web Services 的共享资源池中获得更多的资源。”

Ciancutti 说的“编程式访问”资源，意味着开发人员和运维人员可以通过编程的方式，来访问 Amazon Web Services 上某些公开的管理 API，这相当于为客户提供了一个可以用来配置其虚拟化计算型基础架构的控制器。开发人员可以通过 RESTful API，来构建一些管理和配置其虚拟化基础架构的应用程序。

图 1-1 中所示的图层描绘了不同抽象层次上的云服务类型。

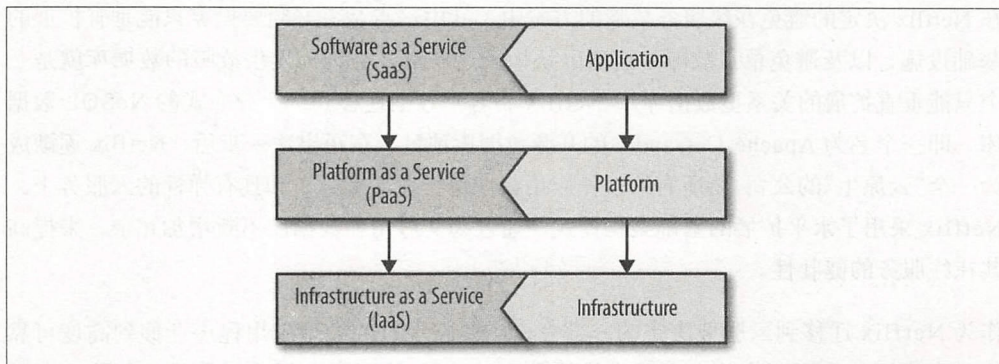


图 1-1 云计算栈





提供管理服务来控制虚拟化的计算型基础架构是云计算的主要概念之一，称为基础架构即服务，通常我们称为 IaaS。

Ciancutti 在同一篇文章中承认，Netflix 不太善于预测客户增长或设备增长。这是采用云原生能力公司背后的核心思考。云原生思想的本质，就是承认无法对所提供服务能力的变化进行可靠的预测。

在 Yury Izrailevsky 2013 年 re:Invent 的演讲中，他说道：“在云端，当流量增加时，我们可以在短短几天内提高系统的处理能力。一开始我们的服务数量可能很少，但是可以随着流量的增加来扩展数量。”

他继续说道：“当我们成为一家全球性的公司时，我们能够利用 Amazon Web Services 遍布全球多个地区的优势，无论客户身处何处，都可以为他们提供一个很好的交互体验。”

受益于 AWS 全球化扩张的规模经济，Netflix 也从中获得了巨大的好处。随着 AWS 将可用区扩展到美国以外的地区，Netflix 只需要使用由 AWS 提供的管理 API，就可以把它的服务扩展到全球各地。

Izrailevsky 引用了企业 IT 云中的一般观点：“当然，云计算是很好的，但对我们来说太贵了。”他对这个论点的回应是：“由于 Netflix 迁移到云上，运维成本下降了 87%。我们现在只支付了数据中心 1/8 的费用。”

Izrailevsky 进一步解释了为什么云为 Netflix 带来了如此大的成本节省：“无须担心容量增长就能够发展业务，这对于我们真的很有帮助。系统可以随着我们业务的发展扩展到相应的规模。”

微服务

我们在本书中会多次提到微服务。虽然这本书不是关于微服务的，但你会发现云原生应用程序和微服务经常是关联在一起的。构建微服务的主要思想之一，就是能够让开发团队根据指定的业务能力来组织自己。这种方法并不新鲜。通过小型的分布式组件来创建一个整体的系统，从而降低交付单个功能到生产环境中的风险，这种工作方式已经存在了很久很久。那为什么现在又重点提到微服务呢？为什么这种架构现在变得流行起来？它与云有关吗？

构建软件一直很难。理论和实践之间的不同，往往会导致过去的理论被新的经验所取代。昨天的技术决策会妨碍你以后对架构做出正确的决策。现在，微服务给了我们一个机会，可以抛弃以前的理念，重新构建一个新的、可能对将来更好的实践理论。



理解一件小事很容易，但是要理解它背后的影响很难。如果我们仔细检查一个设计良好的单体应用程序，我们将看到与今天微服务架构中的模块化、简单性和松耦合等类似的设计。当然，主要的区别是历史。不难理解，如何分层选择不好，一个设计精良的东西也会变得混乱不堪。如果一个人在架构中某个很小的、可替换的单元中做出了错误的选择，那么它的影响会随着时间的推移越来越小。但是，如果他负责的是一个精心设计的大型单体系统中许多独立的模块，那么随着时间推移，他的选择会影响以后其他每个人做出正确的选择。到最后，我们必须妥协——我们不得不在前人的选择上做出稍微更好的决定，而不是按照自己的选择来做。

软件就如同生物一样，随着时间不断在发生变化，也不断受到环境的影响。正因为如此，我们必须也时刻迎接变化，同时坚定推进未来架构的变革。毕竟，未来的架构和敏捷开发一样只是个表面的概念。无论我们现在设计出一个多么完美的系统，我们都无法准确地预测未来功能会发生如何变化，因为我们无法预知未来。市场会决定一个产品的命运。因此，我们只能考虑当下的设计。

微服务在今天不仅仅是一个概念。它的模式和实践依然在不断变化——我们依然无法对它有个清晰的定义。许多垂直行业的企业依然对它持观望态度。

有两股力量在一直影响着架构的变革速度，即微服务和云。云已经大大降低了管理基础设施所需的成本和工作量。今天我们可以使用自助服务工具为应用程序“按需”提供基础设施。从云开始，涌现了大量的创新工具，不断让我们重新思考和重塑我们以前的习惯。昨天我们对于构建软件的认知今天可能就不适用了，并且在大多数情况下，我们也很难分清对错。我们现在发现需要在很多不确定的假设基础上作出艰难的决定：我们的服务器究竟是不是物理的？我们的虚拟机究竟是不是永久的？我们的容器究竟是不是无状态的？我们关于基础设施的假设正不断受到这些新鲜事物的冲击。

拆分单体系统

Netflix 列举了从单体架构迁移到基于云计算的分布式系统架构的两大优势：灵活性和可靠性。

Netflix 的架构在迁移到云原生架构之前，是一个单体的 Java 虚拟机 (JVM) 应用程序。虽然一个大的应用程序在部署上有许多优点，但主要的缺点是，开发团队之间由于需要协调互相的变更，而降低了产品整体的迭代速度。

在构建和运维软件时，越中心化，故障的风险越小，但是需要协调的成本越高。协调也需要时间。因此软件架构越中心化，需要不断协调变更所花费的时间也越多。

单体架构往往也不是非常可靠。当各种组件在同一个虚拟机上共享资源时，其中一个组



件的故障可能会扩散到其他组件，从而导致服务不可用。为了避免单体系统故障的风险，团队之间需要花费更多的精力来进行协调。在一个单独的发布周期中发生的变更越多，出现故障的风险就越大。通过将单体系统拆分成更小、更单一的服务，使得团队在各自独立的发布周期内可以进行更小规模的部署。

Netflix 需要改变的不仅是其构建和运维软件的方式，而且需要改变企业的文化。于是，Netflix 转到了一个新的称为 *DevOps* 的运维模式中。在这个新的运维模式中，与传统项目组的结构不同，每个团队都是一个独立的项目组。在项目组中，每个团队都由专职的运维和产品管理组成。项目团队拥有构建和运维其软件所需的一切权力和职责。

Netflix OSS

随着 Netflix 转型成为云原生的公司，它也开始积极参与开源社区。2010 年底，Netflix 系统与电子商务部工程副总裁 Kevin McEntee 在一篇博客中公布了公司未来在开源社区中的计划。

McEntee 表示：“一个能解决大家共同问题的好的开源项目，它的伟大之处在于它不仅带动了自己的发展，还可以通过持续改进的良性循环，发展很长一段时间。”

在该声明公布后的几年内，Netflix 开源了 50 多个内部项目，每个项目都成为了 *Netflix OSS* 品牌的一部分。

Netflix 的主要员工后来解释了公司为何开源如此多的内部工具。2012 年 7 月，Netflix 云平台工程总监 Ruslan Meshenberg 在公司的技术博客上发表了一篇文章。在这篇名为“Netflix 的开源之路”的博客文章中，他解释了为什么 Netflix 采取如此大胆的方式，开源如此多的内部工具。

Meshenberg 在博客文章中写了关于开源背后的原因：“Netflix 在很早期就使用了云计算平台，我们将所有的流媒体服务都迁移到 AWS 的基础架构之上运行。我们在这个过程中踩了很多坑，遇到和处理过许多的问题、意外和各种限制情况。”

Netflix 对开源社区和技术生态系统的贡献回馈，与微观经济理论背后的规模经济概念紧密相连。“我们已经找到了如何在平台组件和自动化工具中运行的模式，”Meshenberg 说道，“我们受益于其他 AWS 用户采用类似模式的规模效应，并将继续与社区合作来开发整个生态系统。”

随着云计算时代的到来，我们已经看到，这个时代的先驱者不是像 IBM 或者微软这样的科技公司，而是从互联网中诞生的公司。Netflix 和 Amazon 都是在 20 世纪 90 年代末成立的互联网公司。两家公司都从一开始就瞄准了实体竞争对手，通过提供在线服务开展业务。



Netflix 和亚马逊都已经超过了它们的实体对手的估值。随着亚马逊进入云计算市场，它将集体经验和内部工具转变成为了一整套服务。Netflix 在亚马逊的服务的基础上也是如此。这一路走来，Netflix 已经将它如何在 AWS 虚拟化架构服务之上，构建一个云原生公司的经验和工具都开源了出来。这就是规模经济在推动云计算行业革命的最好证明。

在 2015 年初，根据 Netflix 第一季度的收益报告，该公司的估值为 329 亿美元。按照 Netflix 新的估值结果，该公司的价值首次超过了 CBS 网络的价值。

云原生 Java

作为一家成功迁移到云原生的公司，Netflix 为软件行业提供了丰富的经验和知识。本书将采用一些 Netflix 的教程和开源项目，并将它们按照内容分成两类主题：

- 使用 Spring 和 Netflix OSS 构建弹性的分布式系统。
- 通过持续交付来运维 Cloud Foundry 上的云原生应用程序。

我们旅程的第一站，是了解本书在描述构建和运维云原生应用程序时，所使用的一套术语和概念。

十二要素原则

十二要素原则是由 Heroku 云平台的创建者所编写的一套应用程序开发理论。*Twelve-Factor App* 是一个由 Heroku 联合创始人 Adam Wiggins 创建的网站，用于描述利用现代云平台来实际构建 SaaS 应用程序的一份宣言。

在网站 (<http://12factor.net>) 上，该方法论从描述一组用于构建应用程序的核心基础思想开始。

在本章的前面部分，我们讨论了平台为用户所做出的承诺。在表 1-1 中，我们明确提出了按照十二要素原则构建应用程序的价值主张。这些想法进一步被分解成一系列的约束——这 12 个独立的要素将这些核心思想提炼成了一组如何构建应用程序的意见。

表1-1 十二要素程序的核心思想

使用 声明 的方式来搭建自动化环境，最大限度地减少新加入项目的开发人员的时间和成本
与底层操作系统之间建立清晰的约定，在执行环境之间提供最大的 可移植性
适合 部署 在现代的云平台上，无须提供服务器和系统管理工具
最大程度地减少开发环境与生产环境之间的区别，通过 持续部署 获得最大的灵活性
可以在不对工具、架构或开发实践带来重大变动的前提下，进行 水平扩展

表 1-2 中列出的 12 个要素，描述了按照表 1-1 构建应用程序时所使用的约束。12 个要素是可以用于构建云原生应用程序的基本约束条件。由于这些要素涵盖了所有现代云平台实践中的常见问题，所以在开发云原生应用程序时，通常都从构建符合这 12 个要素的应用程序开始。

表1-2 十二要素程序的实践

代码库	一份版本控制下的基准代码库，多份部署
依赖	显式声明和隔离依赖关系
配置	在环境中存储配置
后端服务	把后端服务当作附加资源
构建，发布，运行	严格分离构建和运行阶段
进程	将应用程序作为一个或多个无状态进程执行
端口绑定	通过端口绑定暴露服务
并发	通过进程模型进行扩展
易处理	通过快速启动和正常关机来最大限度地提高健壮性
开发 / 生产环境一致	尽可能保持开发、预发布和生产环境的配置一致
日志	将日志视为事件流
管理进程	将管理任务作为一次性进程运行

除了详细介绍十二要素的网站之外，本书对其中每一点就进行了扩展说明。十二要素方法论现在已经被应用在一些应用程序的开发框架之中，以便帮助开发人员遵守部分甚至全部这十二个要素。

本书通篇将使用十二要素的方法论，来阐述 Spring 项目的某些功能是如何满足这种开发风格的。因此，我们在这里概括介绍一下每一个要素。

代码库

一份版本控制下的基准代码库，多份部署

应用程序的源代码仓库应该只包含一个应用程序，并列出它所依赖的资源清单。对于不同的环境，我们应该不需要重新编译或打包应用程序。每个环境中特有的设置应该与代码无关，如图 1-2 所示。

依赖

显式声明和隔离依赖关系

应该显式地声明应用程序的依赖关系，并且可以通过依赖项管理软件（如 Apache Maven）的仓库，下载任意或所有的依赖关系。

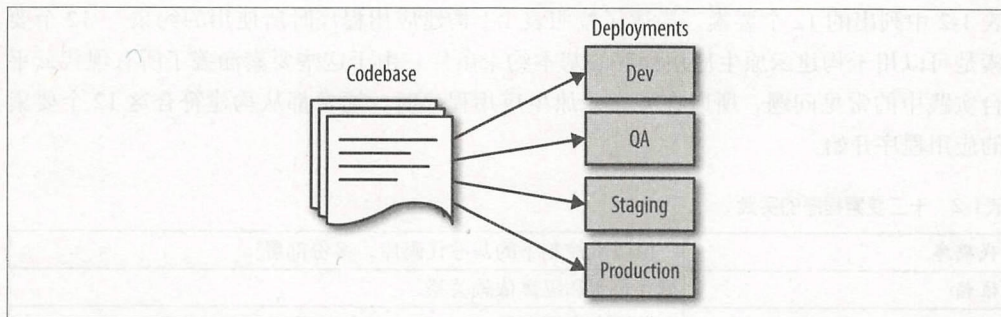


图1-2 构建一次源代码可部署到多个环境中

满足十二要素的程序，不应该依赖于底层系统级包才能够运行。应用程序的所有依赖关系都应该被明确地声明在清单文件中，该文件清晰地列出了每个引用的详细信息。

配置

在环境中存储配置

应用程序的代码应与配置严格分开。应用程序的配置应由各自环境管理。

所依赖服务的连接字符串、密码或者主机名等，应该存储为环境变量，这样易于修改，无须部署配置文件。

应用程序在环境之间的任何差异，应该都认为是一种环境配置，并且应该存储在环境中，而不是应用程序中，如图 1-3 所示。

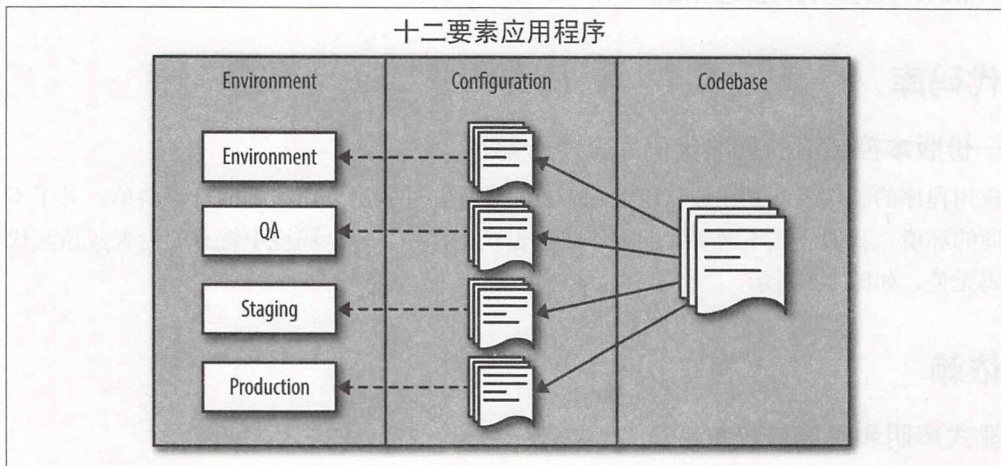


图1-3 在外部存储环境中的特定配置

后端服务

把后端服务当作附加资源

后端服务是十二要素程序正常运行所需要依赖的任何服务。举例说明，后端服务包括数据库、API 驱动的 RESTful Web 服务、SMTP 服务器或 FTP 服务器。

后端服务被认为是应用程序的某种资源。这些资源在运行期间被绑定到应用程序上。十二要素程序应该在无须代码改动的前提下，可以从一个测试环境中的嵌入式 SQL 数据库，切换到预发布环境下的一个独立 MySQL 数据库。

构建、发布、运行

严格分离构建和运行阶段

十二要素程序严格分离了构建、发布和运行几个阶段。

构建阶段

构建阶段将应用程序的源代码编译或打包到一个程序包中。创建的包被称为一次构建物。

发布阶段

发布阶段需要将某次构建与其配置相结合。随后，创建出的发布文件应该可以在某个执行环境中运行。无论是使用版本号还是时间戳，每个版本应该有一个唯一的标识符。每个发布文件都应该被添加到一个目录中，可以通过发布管理工具回滚到之前的发布版本。

运行阶段

运行阶段（通常称为运行时）是指在可执行环境中运行一个指定的应用版本。

通过将这些阶段分成独立的流程，在运行时就完全不可能去更改任何程序代码。更改应用程序代码的唯一方法，就是通过构建阶段来创建一个新的版本，或者回滚到之前部署的某个版本。

进程

将应用程序作为一个或多个无状态进程执行

十二要素程序应该是一种无共享、无状态的架构。应用程序可以依赖的唯一持久化元素是后端服务。提供持久化的后端服务包括数据库或对象存储。应用程序运行时需要的所有资源都应该作为后端服务。无论应用程序是否是无状态的，都需要进行测试，保证应

用程序的执行环节在关闭或启动时都不会有任何数据丢失发生。

十二要素程序不应该在执行环境中的本地文件系统中存储任何有关的状态信息。

端口绑定

通过端口绑定暴露服务

十二要素程序本身是完全独立的，这表示它们不需要在运行时引入一个 Web 服务器，就可以提供对外的 Web 服务能力。每个应用程序通过将自身 HTTP 端口绑定到执行环境中，来对外公开自己的访问。在部署期间，对于来自公开主机的请求，路由层会将请求路由到应用程序的执行环境和绑定的 HTTP 端口来进行处理。



本书的合著者之一 Josh Long，由于在 Java 社区中广泛推广“使用 JAR 包而非 WAR 包”的口号而知名。Josh 使用这个口号来说明，新的 Spring 程序可以在构建出的 JAR 包中嵌入一个 Java 应用服务器，例如 Tomcat。

并发

通过进程模型进行扩展

在需要时，应用程序应该能够通过增加进程或线程来并行执行工作。JVM 应用程序能够使用多个线程自动处理进程中的并发情况。

根据类型不同，应用程序应该能够同时分配多个工作。目前，JVM 的大多数应用程序框架都内置了这个能力。对于一些需要长时间运行的数据处理工作，你应该使用执行器，异步地将多个并发工作分配给可用的线程池。

十二要素程序还必须能够水平扩展，并且能够将请求分发到负载均衡背后的多个程序实例处理。通过确保应用程序是无状态的，你可以增加多个节点水平扩展程序来处理更多的请求。

易处理

通过快速启动和正常关机来最大限度地提高健壮性

十二要素程序需要被设计成易处理的。应用程序应该可以在执行过程中的任意时刻停止，并且能够优雅地释放进程。

应用程序的进程应尽可能地减少启动时间。应用程序应该在几秒钟内启动，并且开始能

够处理接收的请求。短暂的启动时间会降低应用程序进行水平扩展时花费的时间。

如果应用程序的进程启动花费太长时间，可能会在高峰时间导致所有的应用程序实例过载，从而降低系统整体的可用性。通过将应用程序的启动时间减少到几秒钟，新的实例能够更快速地响应不可预测的流量峰值，而不会降低系统的可用性或者性能。

开发 / 生产环境一致

尽可能保持开发、预发布和生产环境的配置一致

十二要素程序应防止开发和生产环境之间存在差异。有三种类型的差异要注意。

时间差异

开发人员希望开发的变更代码能够快速部署到生产环境中。

人员差异

更改代码的开发人员应该密切关注生产环境的部署，并在部署后密切监控应用程序的行为。

工具差异

每个环境应该具备相同的技术和框架，避免因为一些细小的意外差异导致不一致的行为出现。

日志

将日志视为事件流

十二要素程序将写日志看作到标准输出的一个有序的事件流。应用程序不应负责管理自身日志文件的存储。应用程序日志输出的收集和归档工作应该由运行环境来完成。

管理进程

将管理任务作为一次性进程运行

有时，应用程序的开发人员需要运行一次性的管理任务。这类任务可能包括数据库迁移或运行已提交到源代码仓库中的一次性脚本。这些都被认为是管理进程。管理进程应该在应用程序的执行环境中运行，并将脚本提交到代码仓库中，以便保持各个环境之间的一致性。

总结

在本章中，我们研究了一些驱使企业接受某些约束和进行架构迁移的动机。并探讨了一些好的理念，特别是刚刚介绍的十二要素宣言。

训练营：Spring Boot和Cloud Foundry

在本章中，我们将讨论如何使用 Spring Boot 和 Cloud Foundry 来构建云原生的应用程序。

什么是 Spring Boot

当我们说一个应用程序是云原生程序的时候，这意味着它是被设计来在一个云计算环境中运行的。Spring Boot 提供了一种方法，只需要最少的安装时间，就可以创建一个可运行在生产环境中的 Spring 应用程序。创建 Spring Boot 项目的主要目标，是用户应该能够使用 Spring 快速启动和运行程序。

Spring Boot 还从自己的角度来看待 Spring 平台和第三方库，它提供了一组对于所有 Spring 项目都通用的抽象，无须开发人员介入就可以把所有项目的需求连接起来。通过这种方式，Spring Boot 可以在项目需求变化时，非常方便地更换组件。Spring Boot 建立在 Spring 的生态系统和第三方库的基础上，且增加了自己的特点，并建立了一套适合创建云原生应用程序的规范。

Spring Initializr 入门

Spring Initializr 是一个开源项目 (<https://github.com/spring-io/initializr>)，也是 Spring 生态系统中的一个工具，可以帮助你快速生成新的 Spring Boot 应用程序。Pivotal 在 Pivotal Web Services (<http://start.spring.io>) 上运行了一个 Spring Initializr 的实例。它可以生成 Maven 和 Gradle 工程，且包含任意指定的依赖项、一个作为项目启动入口的 Java 类以及一个单元测试模板。

对于单体应用程序，这种成本可能令人望而却步，但是其在项目的整个生命周期内，却可以轻松的分摊整个初始化工作的成本。当你开始向云原生架构迁移时，你会发现需要创建越来越多的应用程序。正因为如此，应该将创建新应用程序的阻力降到最低。Spring Initializr 有助于减少前期的成本。它同时也是一个 Web 应用程序，你可以通过 Web 浏览器和 REST API 来生成新的项目。

例如，你可以使用 curl 命令来生成新的 Spring Boot 项目：

```
curl http://start.spring.io
```

结果如图 2-1 所示。

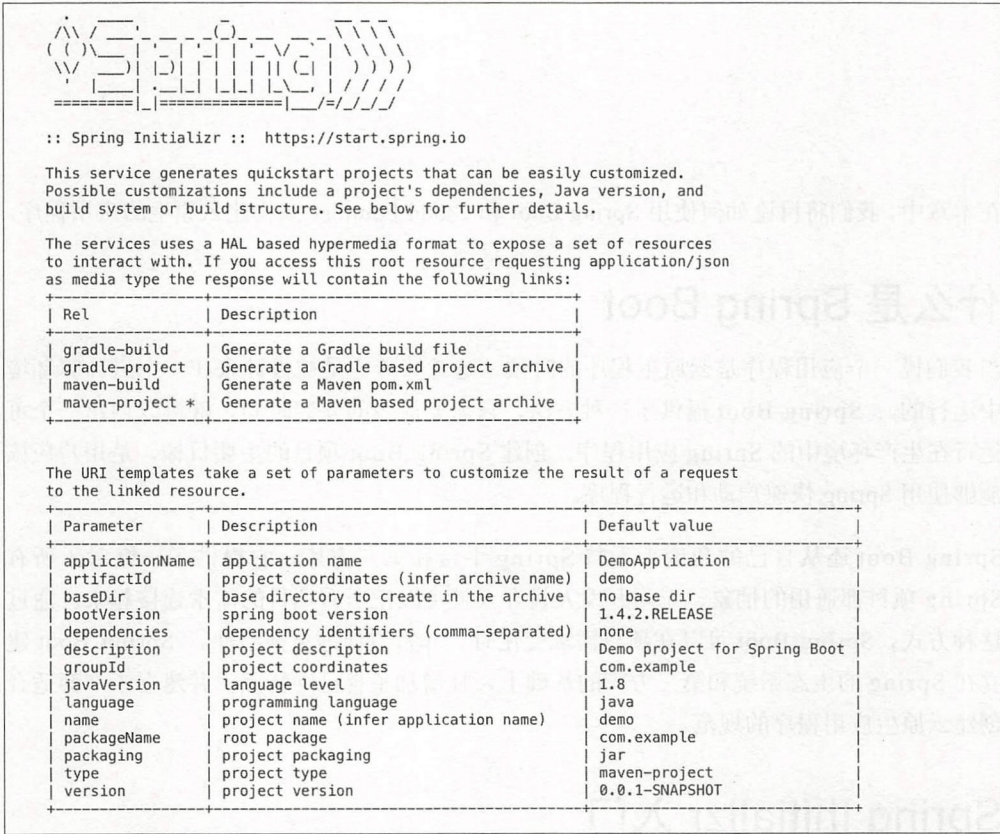


图2-1 通过REST API与Spring Initializr进行交互

或者，可以在浏览器中使用 Spring Initializr (<http://start.spring.io>)，如图 2-2 所示。

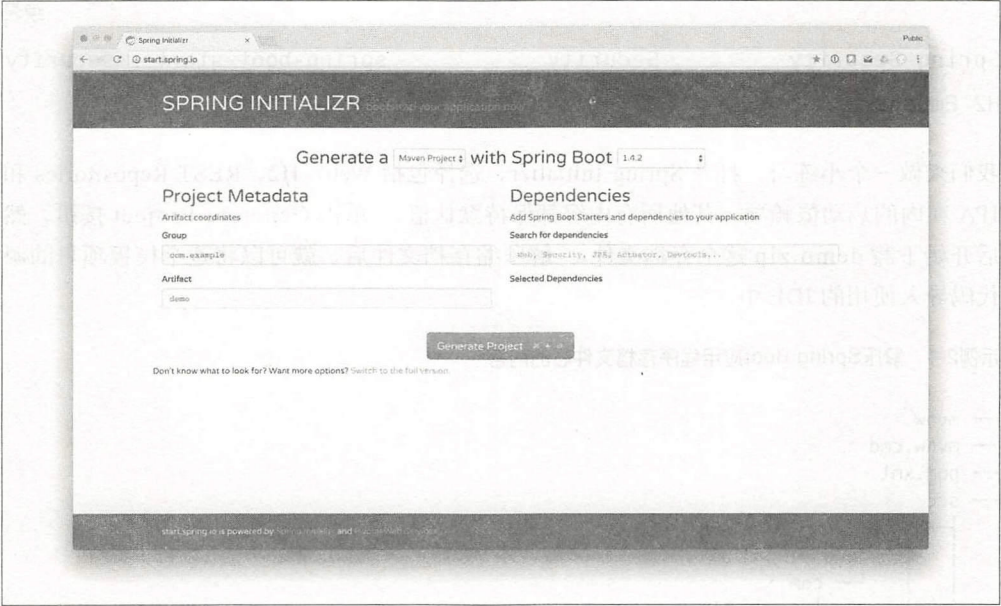


图2-2 Spring Initializr的网站

现在，假设我们要创建一个简单的、与 SQL 数据库（H2）进行交互的 RESTful Web 服务。我们需要使用 Spring 生态中相关的库，其中包括 Spring MVC、Spring Data JPA 和 Spring Data REST。

要找到这些代码库，可以在搜索框中输入依赖项的名称，或单击“切换到完整版本”的按钮，然后手动选择所需依赖库的复选框。它们中的大多数被称为启动项目依赖项。

启动依赖项是一种特有的加载库，它会自动将一组由 Spring 其他项目组成的基本功能，注入新的 Spring Boot 应用程序中。

如果我们需要从头开始搭建 Spring Boot 应用程序，我们必须清楚依赖项之间的传递关系。因此，可能会遇到依赖项之间共享的第三方库存在版本冲突的问题。但是 Spring Boot 可以处理冲突的传递依赖关系，因此你只需要指定 Spring Boot 父项目的一个版本即可，这样就可以保证任何在类路径上的 Spring Boot 启动依赖项，都会使用相互兼容的版本。

表2-1 一个典型Spring Boot应用程序所用到的Spring Boot启动项示例

Spring Project	Starter Projects	Maven artifactId
Spring Data JPA	JPA	spring-boot-starter-data-jpa
Spring Data REST	REST Repositories	spring-boot-starter-data-rest
Spring Framework (MVC)	Web	spring-boot-starter-web

Spring Security	Security	spring-boot-starter-security
H2 Embedded SQL DB	H2	h2

我们来做一个小练习。打开 Spring Initializr，选择包括 Web、H2、REST Repositories 和 JPA 在内的启动依赖项，其他所有内容都保持默认值。单击 Generate Project 按钮，然后开始下载 demo.zip 这个存档文件。解压缩存档文件后，就可以将这个模板项目的源代码导入使用的 IDE 中。

示例2-1 解压Spring Boot应用程序存档文件后的内容

```

.
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   ├── com
    │   │   │   └── example
    │   │   │       └── DemoServiceApplication.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        ├── java
        │   ├── com
        │   │   └── example
        │   │       └── DemoServiceApplicationTests.java

```

在示例 2-1 中，我们可以看到所生成的应用程序的目录结构。Spring Initializr 会同时在项目中提供一个包装器脚本，或者 Gradle 的包装器 (gradlew)，或者 Maven 的包装器 (来自 Maven 的 wrapper 项目) mvnw。可以使用这个包装器来构建和运行项目。包装器在第一次运行时会自动下载构建工具的指定版本。因为 Maven 或 Gradle 受到版本控制，所以所有后续用户都可重复进行构建。不会有人尝试用不兼容版本的 Maven 或 Gradle 来构建代码。这也大大简化了持续交付的工作：开发使用的构建环境与持续集成环境中使用的完全相同。

以下命令将执行 Maven 项目的清理安装，下载并缓存 pom.xml 中指定的依赖项，并将构建的 .jar 工件安装到本地的 Maven 仓库 (通常在 \$HOME/.m2/repository/* 下) 中：

```
$ ./mvnw clean install
```

如果要从命令行来运行 Spring Boot 应用程序，可使用自带的 Spring Boot Maven 插件，

它已经被自动配置在了生成的 pom.xml 中：

```
$ ./mvnw spring-boot:run
```

这时应用程序应该开始启动，随后可以通过 `http://localhost:8080` 进行访问。不用担心，到这个时候，还没有什么特别有趣的事情发生。

用你习惯的文本编辑器（emacs、vi、TextMate、Sublime、Atom、Notepad.exe 等都可以），打开项目的 pom.xml 文件（见示例 2-2）。

示例2-2 demo-service项目pom.xml文件中的依赖项部分

```
<dependencies>
```

❶

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

❷

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

❸

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

❹

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

❺

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

```
</dependencies>
```

- ❶ spring-boot-starter-data-jpa 能够通过 Java ORM（对象关系映射）规范和 JPA（Java Persistence API）来持久化 Java 对象，开发效率很高。其中包括了 JPA 的规范类型、基本的 SQL Java 数据库连接（JDBC）和 JPA 对 Spring 的支持、作为底层实

现的 Hibernate、Spring Data JPA 以及 Spring Data REST。

- ❷ `spring-boot-starter-data-rest` 能够让从 Spring Data repository 定义中导出超媒体感知的 REST 服务，变得非常简单。
- ❸ `spring-boot-starter-web` 提供了使用 Spring 构建 REST 应用程序所需的一切。它引入了 JSON 和 XML 序列化支持、文件上传支持、嵌入式 Web 容器（默认是最新版本的 Apache Tomcat）、认证支持以及 Servlet API 等。这里其实重复了，因为 `spring-boot-starter-data-rest` 会自动引入这个依赖。为了清楚起见，我们这里还是着重说明一下。
- ❹ `h2` 是一个内存式的嵌入式 SQL 数据库。如果 Spring Boot 在类路径中检测到类似 H2、Derby 或 HSQL 的嵌入式数据库，并检测到你还没有配置 `javax.sql.DataSource`，那么它将为你自动配置一个。当应用程序启动时，嵌入式的 `DataSource` 也会自动启动，当应用程序关闭时它也会自动销毁（包括其中所有内容）。
- ❺ `spring-boot-starter-test` 引入了编写 mock 和集成测试所需的所有依赖，以及 Spring MVC 测试框架。Spring Boot 默认项目需要测试支持，因此会默认添加该依赖项。

因为在父构建中已经指定了所有这些依赖关系的版本，所以在一般的 Spring Boot 项目中，只有 Spring Boot 本身的版本号需要明确指定。一旦 Spring Boot 有新的版本可用，只需将其版本号更换为新的版本号即可，所有相关库的版本号都会相应进行更新。

接下来，使用熟悉的文本编辑器，打开应用程序的入口类 `demo/src/main/java/com/example/DemoApplication.java`，并将其替换为示例 2-3 所示的代码。

示例2-3 一个含有JPA实体Cat的Spring Boot应用程序

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
```

❶

```
@SpringBootApplication
public class DemoApplication {
```

```
    public static void main(String[] args) {
```

❷

```
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

❸

```
@Entity
```

```
class Cat {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private Long id;
```

```
    private String name;
```

```
    Cat() {
```

```
    }
```

```
    public Cat(String name) {
```

```
        this.name = name;
```

```
    }
```

```
    @Override
```

```
    public String toString() {
```

```
        return "Cat{" + "id=" + id + ", name='" + name + '\'' + '}';
```

```
    }
```

```
    public Long getId() {
```

```
        return id;
```

```
    }
```

```
    public String getName() {
```

```
        return name;
```

```
    }
```

```
}
```

❹

```
@RepositoryRestResource
```

```
interface CatRepository extends JpaRepository<Cat, Long> {
```

```
}
```

❶ 用注解标记该类为 Spring Boot 应用程序。

❷ 启动 Spring Boot 应用程序。

❸ 一个简单的 JPA 实体模型 Cat。

- ④ 一个已经暴露为 REST API 的 Spring Data JPA repository（它能够处理所有常见的增删改查操作）。

这段代码应该可以工作，但是只有我们进行测试后才能确定！如果有测试用例，可以为它建立一个基准的工作状态，然后在基准质量上进行改进。所以，打开测试类 demo/src/test/java/com/example/DemoApplicationTests.java，并将其替换为示例 2-4 所示的代码。

示例2-4 应用程序中的一个集成测试类

```
package com.example;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;

//@formatter:off
import org.springframework.boot.test.autoconfigure.web
    .servlet.AutoConfigureMockMvc;
//@formatter:on
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import java.util.stream.Stream;

import static org.junit.Assert.assertTrue;
//@formatter:off
import static org.springframework.test.web.servlet
    .request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet
    .result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet
    .result.MockMvcResultMatchers.status;
//@formatter:on
❶

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
public class DemoApplicationTests {

    ❷
    @Autowired
    private MockMvc mvc;

    ❸
    @Autowired
    private CatRepository catRepository;
```

```

4
@Before
public void before() throws Exception {
    Stream.of("Felix", "Garfield", "Whiskers").forEach(
        n -> catRepository.save(new Cat(n));
    }
}

5
@Test
public void catsReflectedInRead() throws Exception {
    MediaType halJson = MediaType
        .parseMediaType("application/hal+json;charset=UTF-8");
    this.mvc
        .perform(get("/cats"))
        .andExpect(status().isOk())
        .andExpect(content().contentType(halJson))
        .andExpect(
            mvcResult -> {
                String contentAsString = mvcResult.getResponse().getContentAsString();
                assertTrue(contentAsString.split("totalElements")[1].split(":")[1].trim()
                    .split(",")[0].equals("3"));
            }
        );
}
}

```

- ❶ 这个单元测试使用了 Spring 框架中的 test runner。我们对它进行一些配置后，可以把它也用于 Spring Boot 的测试中，用来启动一个模拟的 Web 应用程序。
- ❷ 注入一个 Spring MVC 的 MockMvc 客户端，来测试调用 REST 接口的结果。
- ❸ 我们可以引用 Spring 上下文中的任意 bean，包括 CatRepository。
- ❹ 在数据库中插入一些示例数据。
- ❺ 调用 /cats 资源的 HTTP GET 端点。



第 4 章将详细介绍有关测试的内容。

现在可以启动应用程序，访问 <http://localhost:8080/cats>，通过 HAL 编码的超媒体 REST API 来操作数据库。如果你运行测试用例，结果应该是成功（绿色）的！

在开始使用 Spring Boot 构建云原生应用程序之前，需要先搭建一个开发环境。



Spring Tool Suite 入门

到目前为止，我们已经用文本编辑器完成了所有工作，但如今大多数开发人员都会使用 IDE。市面上有很多可以选择的 IDE，而且 Spring Boot 对它们支持得很好。如果你还没有一个 Java IDE，可以考虑使用 Spring Tool Suite (STS) (<http://spring.io/tools>)。Spring Tool Suite 是一个基于 Eclipse 的 IDE，但是与你自己从 Eclipse 官方网站 (<http://eclipse>) 下载 Eclipse 基础平台相比，STS 包含了常见的各种开发工具，能够提供更好的 Eclipse 体验。STS 遵循 Eclipse 公共许可证的条款，可以免费下载使用。

STS 将 Spring Initializr 集成到了 IDE 中。实际上，Spring Tool Suite、IntelliJ 旗舰版、NetBeans 和 Spring Initializr Web 程序最后都会调用 Spring Initializr 的 REST API，所以无论你使用哪个 API，结果都是一样的。

你不需要使用任何特殊的 IDE 来开发 Spring 或 Spring Boot 应用程序。笔者已经尝试过在 emacs、Eclipse、Apache Netbeans（无论是否装有 Spring Boot 插件），以及 IntelliJ IDEA 社区版和旗舰版中创建 Spring 程序，没有出现任何问题。需要特别说明的是，Spring Boot 1.x 需要 Java 6 或更高的版本，支持直接编辑 `.properties` 文件，同时支持使用 Maven 或 Gradle 进行构建。对于任何 2010 年以后发布的 IDE 其都可以提供很好的支持。

如果你习惯使用 Eclipse，那么 Spring Tool Suite 提供了很多不错的功能，可以更好地用来创建基于 Spring Boot 的项目：

- 可以访问 STS IDE 中的所有 Spring 指南。
- 可以使用 IDE 中的 Spring Initializr 来生成新的项目。
- 如果你尝试访问某个在类路径中不存在但是在 Spring Boot 的 `-starter` 依赖中存在的类型，STS 能够自动添加该类型。
- *Boot Dashboard* 可以让本地 Spring Boot 应用程序的修改，无缝地与 Cloud Foundry 部署同步。你可以在 IDE 中进一步调试和热部署 Cloud Foundry 上的应用程序。
- STS 可以直接编辑 Spring Boot 的 `.properties` 或 `.yml` 文件，并提供自动提示的功能。
- Spring Tool Suite 是在 Eclipse 主版本发行后，才发布的稳定的 Eclipse 集成版本。

安装 Spring Tool Suite (STS)

下载并安装 Spring Tool Suite (STS) (<http://www.spring.io>)：

- 访问 <https://spring.io/tools/sts>。
- 选择下载 STS。



- 下载、解压缩并运行 STS。

下载、解压缩并运行 STS 程序后，系统将提示你选择一个工作区。选择合适的工作区位置后，单击 OK 按钮。如果你希望在每次运行 STS 时使用相同的工作区，则可以选中“Use this as the default and do not ask again.”选项。选择工作区位置并单击 OK 按钮后，STS IDE 将进行第一次启动（见图 2-3）。

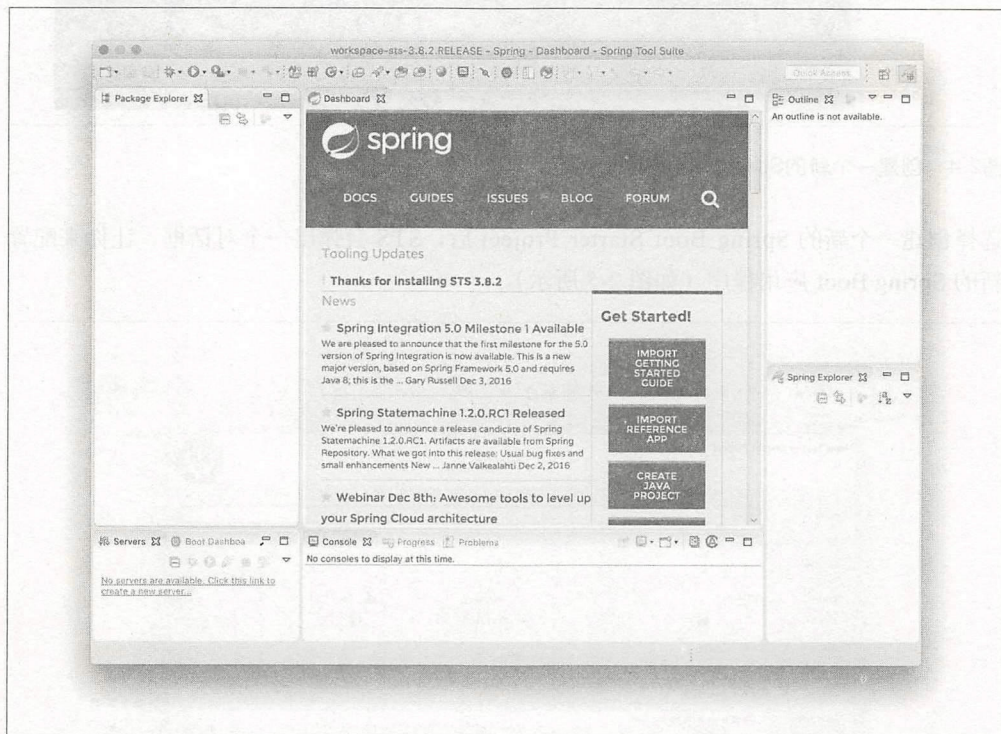


图2-3 STS界面



除了从官网下载之外，还有很多软件包可用于众多的操作系统。例如，如果你在 OS X 或 MacOS Sierra 上使用 Homebrew Cask，则可以使用 Pivotal tap (<https://github.com/pivotal/homebrew-tap>)，通过 `brew cask install sts` 来安装 STS。

使用 Spring Initializr 创建一个新项目

我们可以通过菜单 File → Import → Maven，然后选择 demo 项目下的 pom.xml 文件，直接导入刚刚使用 Spring Initializr 创建的示例程序。但是这里我们使用 STS 来创建第一个 Spring Boot 应用程序。我们会使用 Spring Boot 启动器 (starter) 来创建一个简单的“Hello World” Web 服务。如果要使用 Spring Boot Starter 创建一个新的 Spring Boot 应用程序，



从菜单中选择 File → New → Spring Starter Project，如图 2-4 所示。



图2-4 创建一个新的Spring Boot 启动器项目

选择创建一个新的 Spring Boot Starter Project 后，STS 会弹出一个对话框，让你来配置新的 Spring Boot 应用程序（如图 2-5 所示）。

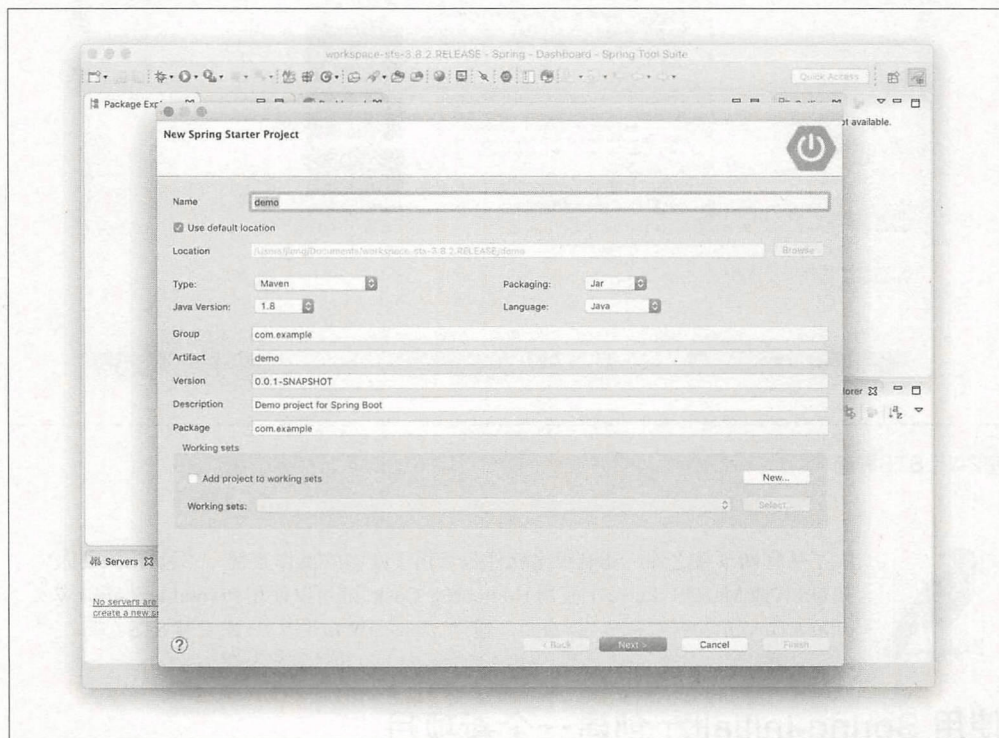


图2-5 配置新的Spring Boot Starter Project

在该对话框中，你可以仔细配置项目的选项，但为了简单起见，我们这里先使用默认值，然后单击 Next 按钮。随后，你会看到一组用来创建新 Spring Boot 应用程序的 Spring



Boot 启动器项目。这里我们选择 Web（如图 2-6 所示）。

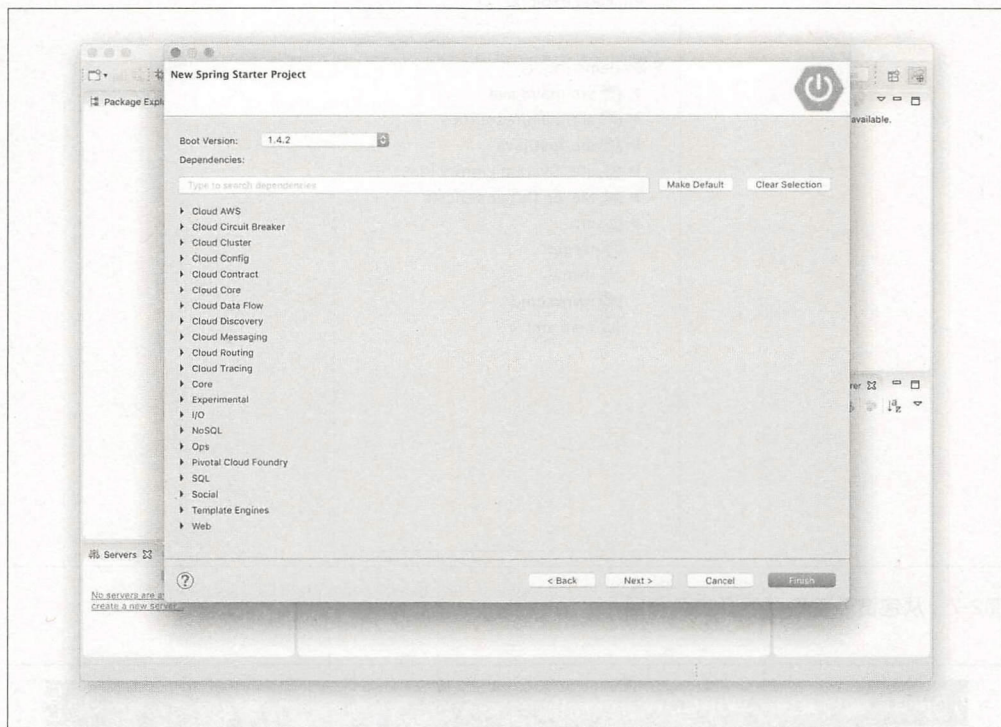


图2-6 选择Spring Boot Starter Project

选择完毕后，单击 Finish 按钮。随后，STS 开始创建 Spring Boot 应用程序，并将其导入 IDE 工作区，于是你就可以在包资源管理器中看见这个项目了。

如果你还没有看到，在 Package Explorer 中展开 *demo [boot]* 节点，并查看项目的内容，如图 2-7 所示。

在展开的项目文件列表中，选择 `src/main/java/com/example/DemoApplication.java`。然后通过菜单 `Run → Run` 来运行应用程序（如图 2-8 所示）。



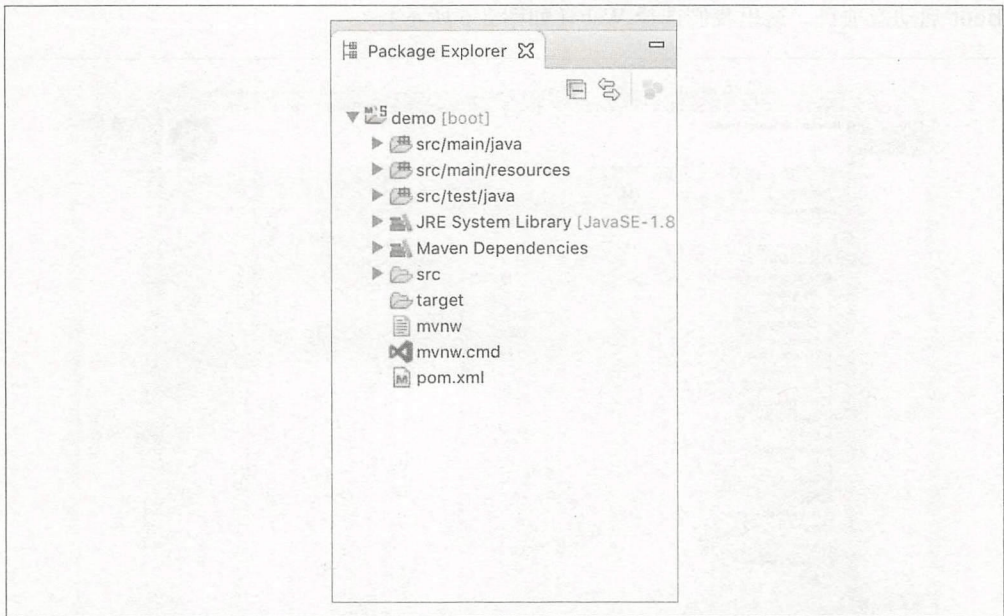


图2-7 从包资源管理器展开演示项目

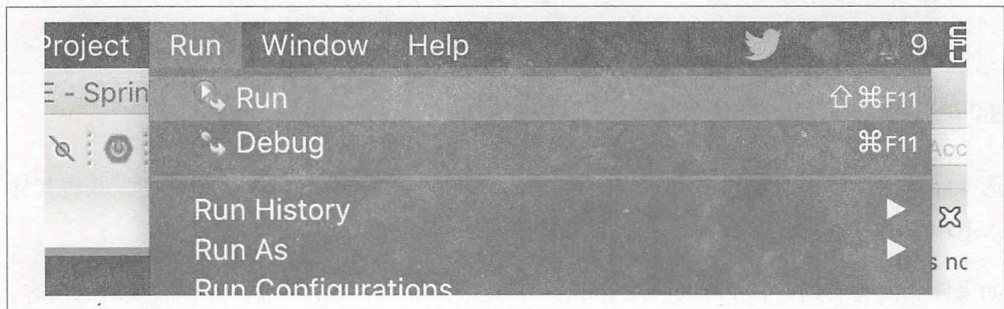


图2-8 运行Spring Boot应用程序

从菜单中选择 Run 选项后，将看到一个 Run As 对话框。选择 Spring Boot App，然后单击 OK 按钮（如图 2-9 所示）。

你的 Spring Boot 应用程序现在应该已经启动了（如图 2-10 所示）。在 STS 的控制台上，应该能看到标志性的 Spring Boot ASCII 艺术字和当前的 Spring Boot 版本。在控制台中你也可以看到 Spring Boot 应用程序输出的日志信息。你可以看到 Spring Boot 在 8080 默认端口上启动了一个嵌入式的 Tomcat 服务器。可以通过 <http://localhost:8080> 来访问这个 Spring Boot Web 服务。



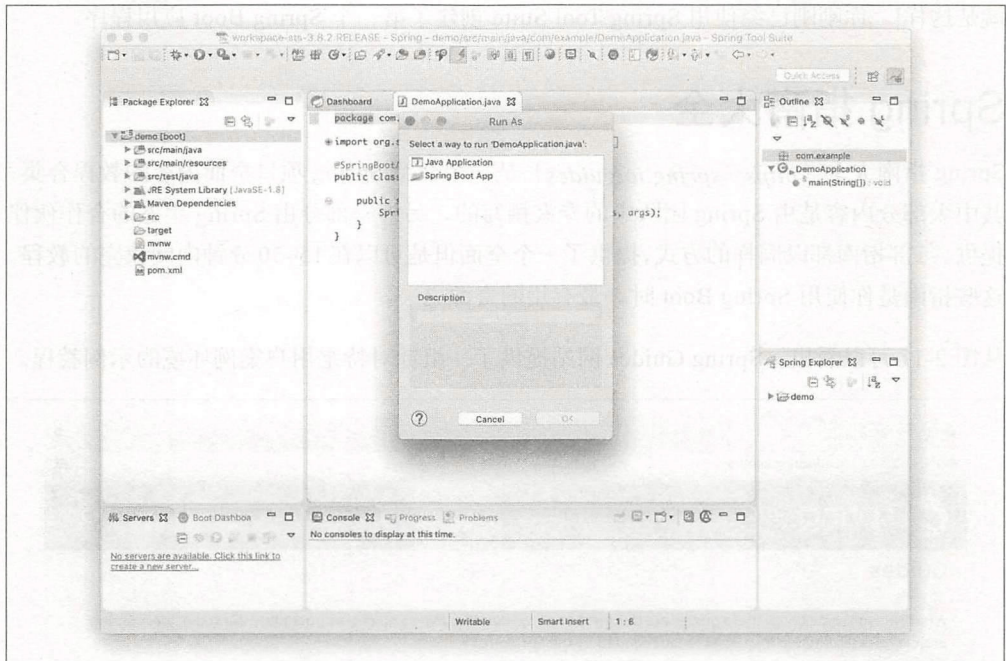


图2-9 选择Spring Boot App并启动应用程序

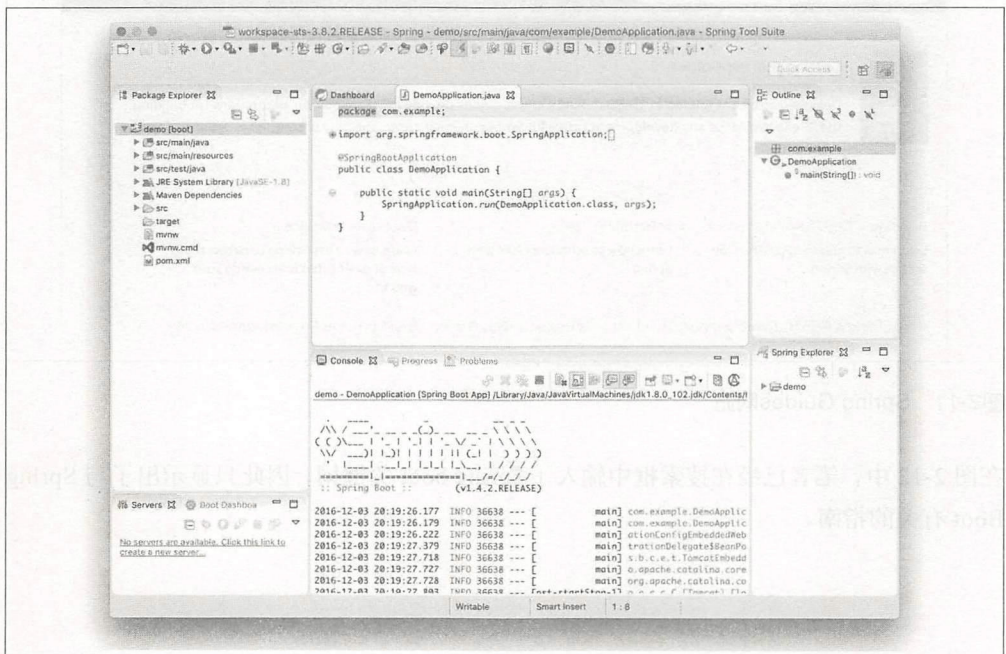


图2-10 查看STS控制台中的Spring Boot的日志输出



就是这样！你刚刚已经使用 Spring Tool Suite 创建了第一个 Spring Boot 应用程序。

Spring 指南大全

Spring 指南大全 (<https://spring.io/guides>) 是一个介绍 Spring 项目全面内容的教程合集，其中大部分内容是由 Spring 团队中的专家撰写的，另外一部分由 Spring 生态的合作伙伴提供。每部指南都以同样的方式，提供了一个全面但是可以在 15~30 分钟内阅读完的教程。这些指南是你使用 Spring Boot 时，最有用的资源之一。

从图 2-11 可以看出，Spring Guides 网站提供了一组针对特定用户案例环境的示例教程。

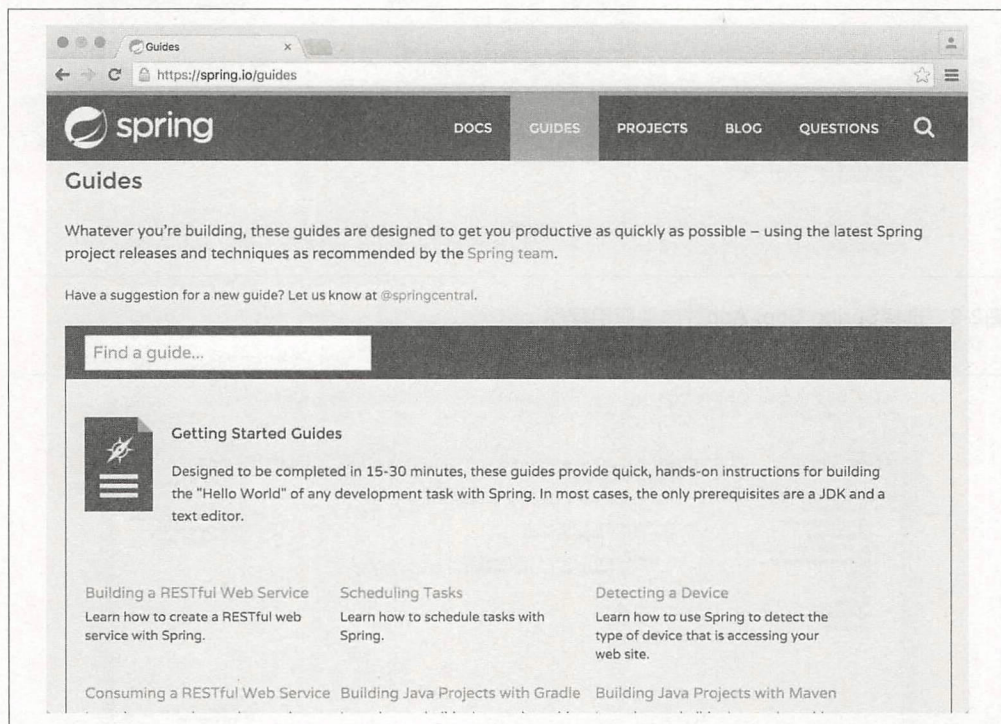


图2-11 Spring Guides网站

在图 2-12 中，笔者已经在搜索框中输入了 spring boot 关键词，因此只显示出了与 Spring Boot 有关的指南。



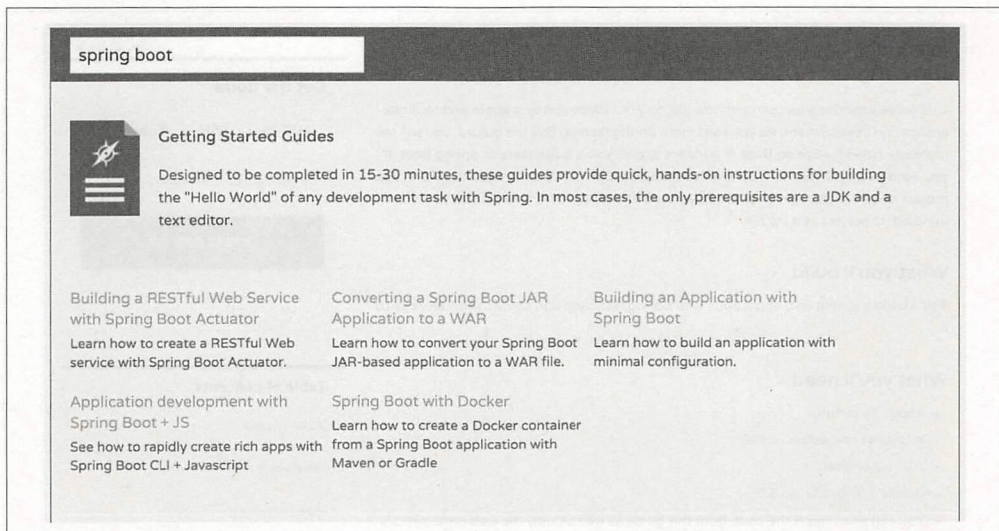


图2-12 查看Spring的相关指南

在每个 Spring 指南中，我们都会发现如下的功能：

- 入门
- 目录
- 你要构建什么
- 你需要什么
- 如何完成本指南
- 练习
- 总结
- 获取代码
- GitHub 仓库的链接

现在我们选择其中一个基础的 Spring Boot 指南“使用 Spring Boot 构建应用程序”(<https://spring.io/guides/gs/spring-boot/>)。

从图 2-13 我们可以了解 Spring 指南的基本结构。每个指南的结构都非常相似，这样可以帮助你更加有效地阅读指南。每个指南都有一个对应的 GitHub 仓库，其中包含三个文件夹：`complete`、`initial` 和 `test`。`complete` 文件夹中包含最终的示例工程，用来检验最终的结果。`initial` 文件夹中包含一个几乎空白的文件系统骨架，可以用它来测试和练习指南中的一些重点、难点。`test` 文件夹用来测试 `complete` 文件夹中代码是否能够正确运行。



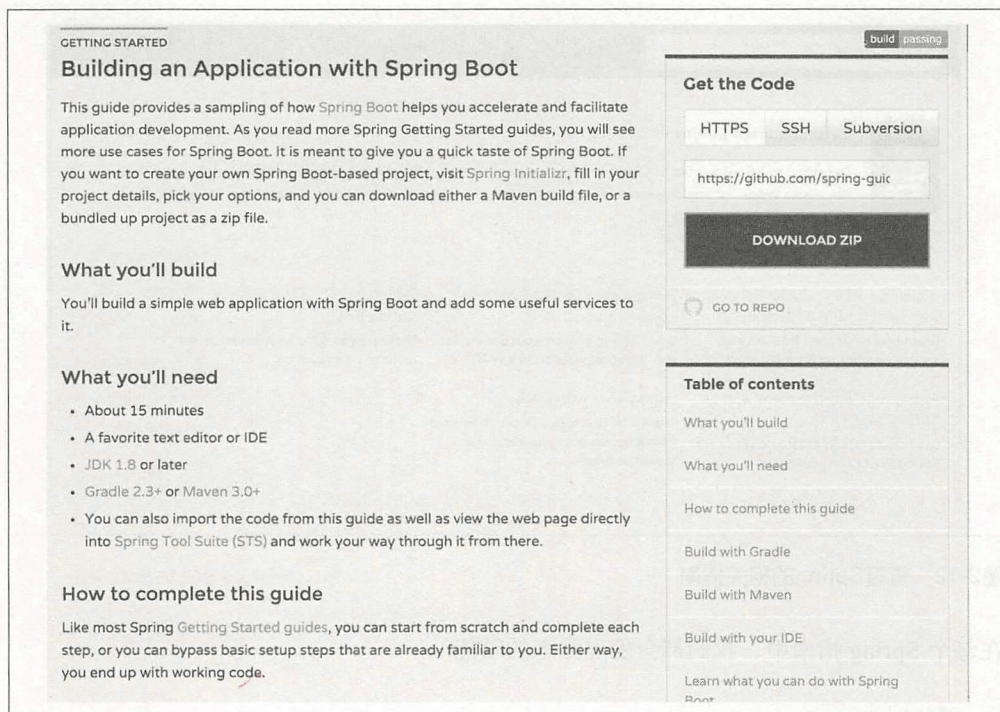


图2-13 使用Spring Boot指南构建应用程序



本书为了让你能更好地理解所讲内容,有时会让你去参考一些扩展知识的指南。如果这样,建议你从 Spring 指南大全中找到最符合需求的指南。

遵循 STS 中的指南

如果你使用的是 Spring Tool Suite,则可以在 IDE 中直接订阅这些指南。在 IDE 中选择菜单 File → New → Import Spring Getting Started Content (如图 2-14 所示)。

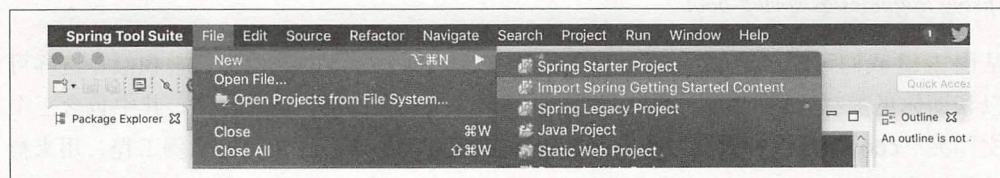


图2-14 选择Import Spring Getting Started Content

这样就可以获取到与 spring.io/guide 上相同的指南目录,如图 2-15 所示。



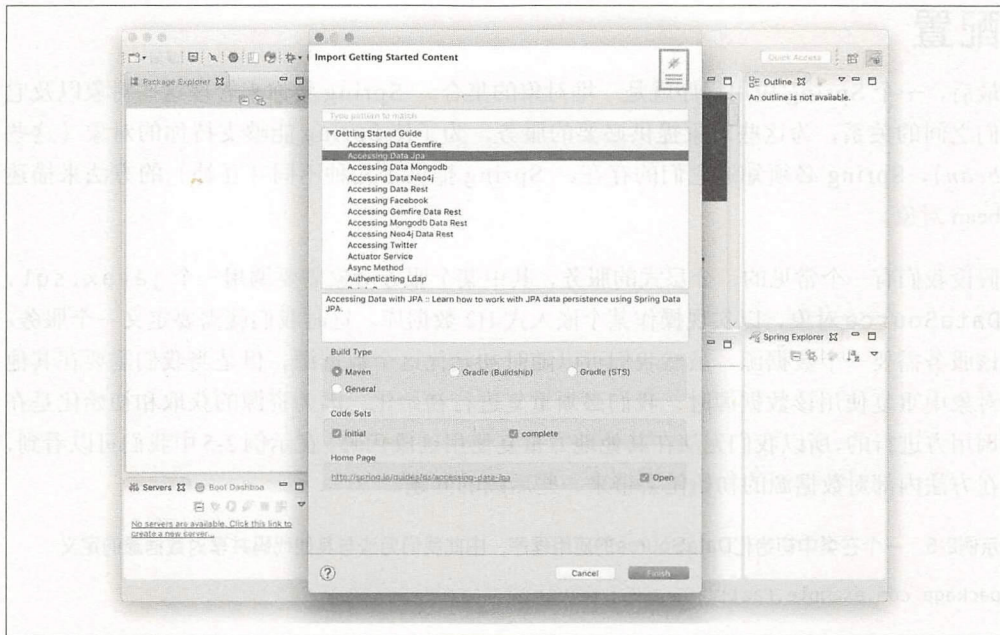


图2-15 从Import Getting Started Content对话框中选择一个指南

选中了某个指南后，该指南及其相关代码就会显示在 IDE 中（如图 2-16 所示）。

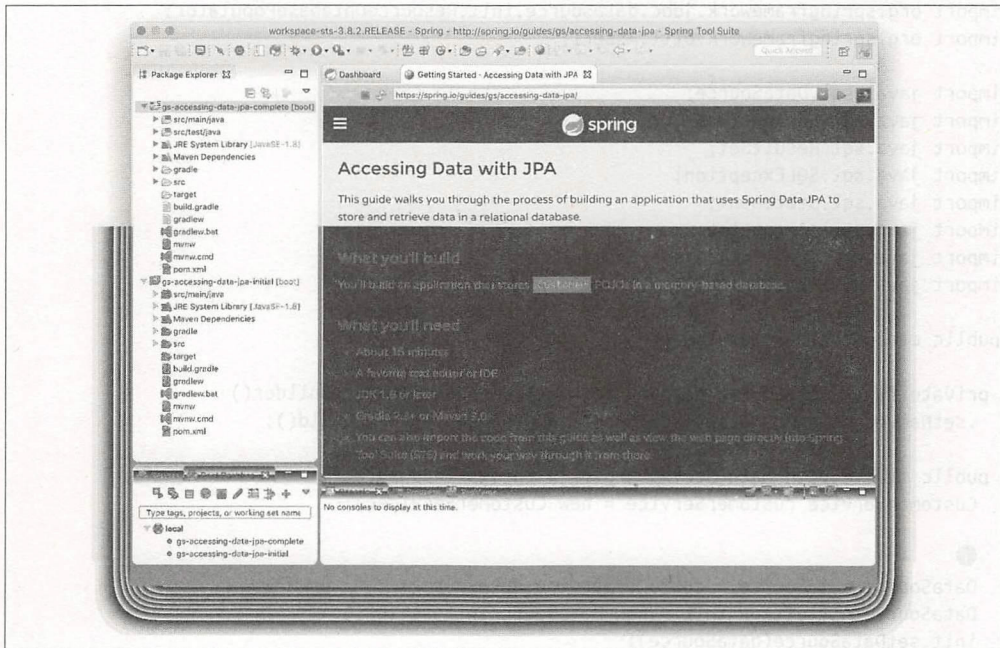


图2-16 IDE会加载选中的指南及其Git项目



配置

最后，一个 Spring 应用程序就是一堆对象的集合。Spring 替你管理这些对象以及它们之间的关系，为这些对象提供必要的服务。为了使 Spring 能够支持你的对象（这些 *bean*），Spring 必须知道它们的存在。Spring 提供了几种不同（互补）的方法来描述 *bean* 对象。

假设我们有一个常见的、分层式的服务，其中某个服务对象需要调用一个 `javax.sql.DataSource` 对象，以实现操作某个嵌入式 H2 数据库。这时我们就需要定义一个服务。该服务需要一个数据源。虽然我们可以随时初始化这个数据源，但是当我们需要在其他对象中重复使用该数据源时，我们必须重复进行初始化。因为资源的获取和初始化是在调用方进行的，所以我们无法在其他地方重复使用这段代码。在示例 2-5 中我们可以看到，在方法内部对数据源的初始化会带来一些后续的问题。

示例 2-5 一个在类中初始化 `DataSource` 的应用程序，由此我们无法与其他代码共享对数据源的定义

```
package com.example.raai;

import com.example.Customer;
import org.springframework.core.io.ClassPathResource;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.jdbc.datasource.init.DataSourceInitializer;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;
import org.springframework.util.Assert;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class CustomerService {

    private final DataSource dataSource = new EmbeddedDatabaseBuilder()
        .setName("customers").setType(EmbeddedDatabaseType.H2).build();

    public static void main(String args[]) throws Throwable {
        CustomerService customerService = new CustomerService();

        ❶
        DataSource dataSource = customerService.dataSource;
        DataSourceInitializer init = new DataSourceInitializer();
        init.setDataSource(dataSource);
        ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
```



```
populator.setScripts(new ClassPathResource("schema.sql"),
    new ClassPathResource("data.sql"));
init.setDatabasePopulator(populator);
init.afterPropertiesSet();
```

②

```
int size = customerService.findAll().size();
Assert.isTrue(size == 2);

}

public Collection<Customer> findAll() {
    List<Customer> customerList = new ArrayList<>();
    try {
        try (Connection c = dataSource.getConnection()) {
            Statement statement = c.createStatement();
            try (ResultSet rs = statement.executeQuery("select * from CUSTOMERS")) {
                while (rs.next()) {
                    customerList.add(new Customer(rs.getLong("ID"), rs.getString("EMAIL")));
                }
            }
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    return customerList;
}
}
```

- ❶ 在这里很难保存数据源，因为它唯一的引用是 `CustomerService` 类中的一个 `private final` 字段。访问该变量的唯一方法是利用 Java 的内部类，这样指定对象的实例才能够访问其他实例的私有变量。
- ❷ 这里我们使用 Spring 本身而不是 JUnit 的类，因为 JUnit 库是用来测试组件的。Spring 框架的 `Assert` 类支持契约式设计，而不仅仅用于单元测试。

由于我们无法插入一个模拟的数据源，也无法在外部访问到这个数据源，因此我们不得不把测试代码也写在这个类里。这种做法不利于测试，而且我们必须在主代码中包含对测试库的依赖。

现在我们使用了一个嵌入式的数据源，它在开发环境和生产环境中是一样的。在一个现实的例子中，与环境有关的配置（例如用户名和主机名），应该是参数化的，否则很容易出现手工错误，将开发环境的数据源配置成了生产环境的参数。

如果我们能够把所有 bean 的定义从代码中拿出来，都集中在一个地方，那样就会清晰很多。我们的代码该如何访问这些集中的定义呢？我们也许可以将它们存储在静态变量



中，但是这样会导致代码中到处都是静态变量，我们又该如何去测试它们呢？我们如何去模拟这些引用？我们也许可以将这些引用存储在某种共享的上下文中，例如 JNDI (Java 命名和目录接口)，但是我们也会遇到同样的问题：如果不模拟出所有的 JNDI，就很难进行这个测试！

为了避免到处都出现这种初始化和调用对象的逻辑代码，我们可以在一个类中创建所有的对象，并建立它们之间的联系。这个原理被称为反转控制 (IoC)。

对象之间的关联跟对象本身无关。为了将它们区分开，我们可以创建只依赖于基础类型和接口的组件代码，而不依赖于特定的实现。这被称为依赖注入。一个不关心特定依赖如何被创建出来的组件，在单元测试时也不会关心它的依赖是不是一个假的（模拟的）对象。

除此之外，我们将对象的关联过程（即对象配置），统一放置在一个单独的 *configuration* 类中。我们来看一下，在示例 2-6 中 Spring 如何通过 Java 配置来做到这一点。



那么 XML 呢？Spring 支持基于 XML 的配置。XML 配置具有很多与 Java 配置相同的优点，它也将各个组件的装配集中到了一个地方。虽然 Spring Boot 依然支持 XML 配置，但是它更适合使用 Java 配置。在本书中，我们不会使用基于 XML 的配置。

示例2-6 一个从各处代码中提取出bean定义的配置类

```
package com.example.javaconfig;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
```

```
import javax.sql.DataSource;
```

①

```
@Configuration
```

```
public class ApplicationConfiguration {
```

②

```
@Bean(destroyMethod = "shutdown")
```

```
DataSource dataSource() {
```

```
    return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
        .setName("customers").build();
```

```
}
```

③

```
@Bean
```

```
CustomerService customerService(DataSource dataSource) {
```

```

    return new CustomerService(dataSource);
}
}

```

- ① 这个类是一个 Spring 的 `@Configuration` 类，它告诉 Spring，它可以找到这些对象的定义并将它们关联起来。
- ② 我们把对 `DataSource` 的定义提取为一个 bean 定义。这样，其他任何 Spring 组件都可以看到并调用这个 `DataSource` 的单个实例。

如果 10 个 Spring 组件都依赖于 `DataSource`，那么默认它们访问的都是内存中的同一个实例。这与 Spring 的作用域概念有关。默认情况下，Spring 中的 bean 是单例的。

- ③ 通过 Spring 来注册 `CustomerService` 实例，并告诉 Spring 在上下文中所有已注册的所有 bean 中，找到参数指定的 `DataSource` 类型的 bean。

返回 `CustomerService` 类，并删除显式创建 `DataSource` 的代码逻辑（如示例 2-7 所示）。

示例2-7 一个更简洁的 `CustomerService` 类，其中资源初始化和获取逻辑都已经被提取到其他地方了

```
package com.example.javaconfig;
```

```
import com.example.Customer;
```

```

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

```

```
public class CustomerService {
```

```
    private final DataSource dataSource;
```

```

    ①
    public CustomerService(DataSource dataSource) {
        this.dataSource = dataSource;
    }

```

```

    public Collection<Customer> findAll() {
        List<Customer> customerList = new ArrayList<>();
        try {
            try (Connection c = dataSource.getConnection()) {
                Statement statement = c.createStatement();
                try (ResultSet rs = statement.executeQuery("select * from CUSTOMERS")) {
                    while (rs.next()) {

```



```

        customerList.add(new Customer(rs.getLong("ID"), rs.getString("EMAIL")));
    }
}
}
}
catch (SQLException e) {
    throw new RuntimeException(e);
}
}
return customerList;
}
}
}

```

- ❶ **CustomerService** 类的定义明显简单了很多，因为它现在仅仅只依赖于 **DataSource** 对象。我们将它的职责限制在了一个更合理的范围内：只负责与 **dataSource** 进行交互，而不是定义 **dataSource** 本身。

配置是显式声明的,但也会有一点繁冗。如果我们愿意,可以让 Spring 为我们做很多事情。为什么我们自己要来做这些重复的工作呢?我们可以使用 Spring 的原型注解来标记自己的组件,并让 Spring 根据约定来实例化我们的对象。

我们回到 **ApplicationConfiguration** 类,通过使用组件扫描让 Spring 来发现我们的原型组件(如示例 2-8 所示)。我们不再需要明确描述如何去构造一个 **CustomerService** bean,所以也可以删除该定义。除了换用 **@Component** 注解之外,**CustomerService** 类与以前完全相同。

示例2-8 将DataSource配置提取到一个单独的配置类中

```

package com.example.componentscan;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import javax.sql.DataSource;

@Configuration
@ComponentScan
❶ public class ApplicationConfiguration {

    @Bean(destroyMethod = "shutdown")
    DataSource dataSource() {
        return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
            .setName("customers").build();
    }
}

```

- ❶ 这个注解告诉 Spring 扫描当前的包（或下级的包）并查找所有使用 `@Component` 等原型注解的对象，然后把这些 bean 注册到应用程序的上下文中。这个注解和其他注解也都被标记了 `@Component` 注解，它就像标签一样被 Spring 作为一种记号。Spring 会感知到它们标记在某些组件上，并为它们创建一个新的对象实例。它默认调用无参数的构造函数，或者当它的所有参数在应用程序上下文中都有对应的引用时，它才会调用含有参数的构造函数。Spring 在 `@Configuration` 类的基础上，通过注解提供了很多其他的服务。

因为示例中的代码直接使用了数据源，为了获得一个简单的结果，不得不编写了大量冗余的 JDBC 模板代码。依赖注入是一个强大的工具，但是它不是 Spring 最吸引人的地方。我们将使用 Spring 最令人瞩目的功能之一——可移植服务抽象来简化与数据源的交互。我们将摒弃冗余的 JDBC 代码，而使用 Spring 框架的 `JdbcTemplate` 来完成这个功能（如示例 2-9 所示）。

示例2-9 使用JdbcTemplate来代替底层的JDBC API

```
package com.example.psa;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
```

```
import javax.sql.DataSource;
```

```
@Configuration
```

```
@ComponentScan
```

```
public class ApplicationConfiguration {
```

```
    @Bean(destroyMethod = "shutdown")
```

```
    DataSource dataSource() {
```

```
        return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.H2)
            .setName("customers").build();
    }
```

```
❶
```

```
    @Bean
```

```
    JdbcTemplate jdbcTemplate(DataSource dataSource) {
```

```
        return new JdbcTemplate(dataSource);
    }
```

```
}
```

- ❶ `JdbcTemplate` 是 Spring 体系中对模板模式的众多实现之一。它提供了很多方便的工具类方法，只需要一行代码即可使用 JDBC。它能够处理资源初始化、获取、释放、异常处理等工作，以便我们可以专注于更重要的工作。

借助于 `JdbcTemplate`，修改后的 `CustomerService` 类变得更加简洁了（如示例 2-10 所示）。

示例2-10 一个更加简单的 `CustomerService` 类

```
package com.example.psa;

import com.example.Customer;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Component;

import java.util.Collection;

@Component
public class CustomerService {

    private final JdbcTemplate jdbcTemplate;

    public CustomerService(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public Collection<Customer> findAll() {
        ❶
        RowMapper<Customer> rowMapper = (rs, i) -> new Customer(rs.getLong("ID"),
            rs.getString("EMAIL"));
        ❷
        return this.jdbcTemplate.query("select * from CUSTOMERS ", rowMapper);
    }
}
```

❶ `query` 方法有很多重载方法，其中一个需要一个 `RowMapper` 对象。它是一个回调对象，Spring 会对每个返回的结果调用该方法，这样就可以将从数据库返回的对象映射到系统中的域对象。`RowMapper` 接口也很适合使用 Java 8 lambda 表达式！

❷ 完成查询只需要一行代码。代码更简单了！

因为我们在 bean 配置中统一控制 bean 的装配，所以可以替换（或注入）bean 的不同实现。甚至，我们可以在所有 bean 没有任何感知的情况下，替换掉所有已经注入的实现。假设我们想记录调用所有方法所花费的时间，我们可以创建一个类，继承现有的 `CustomerService` 类，然后在重写的方法中，在调用 `super()` 方法的前后插入记录日志的功能。虽然日志功能是一个横切关注点，但是为了将其注入对象层次的行为中，我们必须重写所有方法。

在理想情况下，我们不需要绕这么多弯来解决对象切面的问题。像 Java 这种只支持单继承的语言，没有办法对任意对象做到这一点。而 Spring 支持了另一种方案：面向切面

的编程（AOP）。AOP 是一个比 Spring 更大的话题，但 Spring 为 Spring 的对象提供了一个非常适合的 AOP 子集。Spring 的 AOP 核心是切面的概念，用来表示进行切面的行为。而切点表示应用切面时应该匹配的模式。在 Spring 提供的全功能的切点语言中，切点中的模式只是其中一部分。切点语言允许你描述 Spring 程序中对象的方法调用。假设，我们现在想在 CustomerService 类中，创建一个能够匹配所有方法调用的切面，并插入记录方法调用时间的日志。

我们需要将 @EnableAspectJAutoProxy 添加到 ApplicationConfiguration @Configuration 类来激活 Spring 的 AOP 功能。然后，只需将切面功能提取到一个单独的、标注 @Aspect 注解的对象中（如示例 2-11 所示）。

示例2-11 将横切关注点转变为一个Spring的AOP切面

```
package com.example.aop;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;
```

```
import java.time.LocalDateTime;
```

```
@Component
```

```
@Aspect
```

①

```
public class LoggingAroundAspect {
    private Log log = LogFactory.getLog(getClass());
```

②

```
@Around("execution(* com.example.aop.CustomerService.*(..))")
public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
    LocalDateTime start = LocalDateTime.now();
```

```
    Throwable toThrow = null;
    Object returnValue = null;
```

③

```
    try {
        returnValue = joinPoint.proceed();
    }
    catch (Throwable t) {
        toThrow = t;
    }
```

```
    LocalDateTime stop = LocalDateTime.now();
```

```
    log.info("starting @ " + start.toString());
    log.info("finishing @ " + stop.toString() + " with duration "
```



```

+ stop.minusNanos(start.getNano()).getNano());

④
if (null != toThrow)
    throw toThrow;

⑤
return returnValue;
}
}

```

- ❶ 将这个 bean 标记为一个切面。
- ❷ 这段代码声明了，对于任何匹配了 `@Around` 中切点表达式的方法，这段方法都能够在其调用的前后执行。虽然还有许多其他的注解，但当前这个已经足够强大。除此之外，你可以了解一下 Spring 对 AspectJ 的支持。
- ❸ 当匹配该切入点的方法被调用时，首先会调用我们的切面，同时传递一个在当前方法调用上的 `ProceedingJoinPoint` 句柄对象。我们可以选择查看方法的执行情况、继续执行或者跳过它，等等。这个切面会在执行方法调用前后记录日志。
- ❹ 如果抛出一个异常，Spring 会将它缓存并稍后重新抛出。
- ❺ 如果记录了一个返回值，它也会被同时返回（假设没有抛出异常）。

如果需要，可以直接使用 AOP，但是 Spring 为我们提供了常见的、重要的横切面关注点。其中一个例子就是声明式事务管理。在我们的示例中有一个只读方法，如果我们打算增加另一个业务方法，例如在单个服务调用中执行多次数据库变更，那么需要保证这些变更都发生在一个单独的工作单元中。要么每次与状态型资源（数据源）的交互都成功，要么都不成功。

我们不想让系统处于不一致的状态。一个使用横切面关注点的理想示例是，我们可能会在一个业务服务中，在调用每个方法之前使用 AOP 来开启一个事务，并在调用完成时提交（或回滚）该事务。虽然我们可以这样做，但是幸运的是，Spring 的声明式事务已经帮我们做到了这一点。我们不需要再编写底层的 AOP 代码来完成类似工作。我们只需要给一个配置类添加 `@EnableTransactionManagement` 注解，然后使用 `@Transactional` 注解来确定业务服务的事务边界。

有了一个服务层之后，下一步应该就是去构建一个 Web 应用程序。可以使用 Spring MVC 来创建 REST 接口。为此，我们需要配置 Spring MVC 本身，将其部署到一个兼容 Servlet 的应用程序服务器中，然后配置应用程序服务器与 Servlet API 的交互。在可以进行下一步，

以及实现一个简单的 Web 应用程序和 REST 接口之前，还有很多工作需要完成！

这里我们直接使用了 JDBC，但是本可以选择使用 ORM 层。这会的事情变得更加复杂。虽然每一次只是更加复杂一点，但是积累起来就会超出我们的处理能力。

这时我们就需要 Spring Boot 及其自动配置的能力。

在本章的第一个例子中，我们创建了一个包含一个接口、一个 JPA 实体、几个注解和一个 `public static void main` 入口类的应用程序，仅此而已！一眨眼，Spring Boot 就被启动起来。然后就在 `http://localhost:8080/cats` 上运行了一个 REST API。这个应用程序能够通过 Spring Data JPA repository 来操作 JPA 实体。我们几乎没有写任何的代码就可以做到这些，实在是太神奇了！

如果你对 Spring 很熟悉，那么你一定意识到，我们正在使用 Spring 框架和它对 JPA 的强大支持。Spring Data JPA 被用来配置一种声明式的、基于接口的资源。Spring MVC 和 Spring Data REST 用来提供基于 HTTP 的 REST API，即使这样，你也会好奇 Web 服务器本身从何而来。每个模块都需要一些配置，通常配置项不多，但肯定比现在的要多。默认情况下，需要使用一个注释来指定它们默认的行为。

在历史上，Spring 曾经选择过对外暴露配置信息。它允许通过一个配置面板 (*Configuration Plane*) 来改变应用程序的行为。在 Spring Boot 中，最先考虑的是提供一个默认合理的配置，但也能够较容易地重新配置。这是对约定优于配置原则的全面支持。最终，按照 Spring Boot 自动配置的优先级，首先使用你手工指定的配置，其次使用你注册的同一个注解和 bean，再次使用通用的默认配置。

Spring 支持服务加载器的概念，能够支持在不改变 Spring 本身的前提下，向应用程序注册自定义的功能。服务加载器是 Java 配置类名称和类型的一组映射，可在加载后供 Spring 应用程序加载和使用。自定义功能通过 `META-INF/spring.factories` 文件进行注册。Spring Boot 会在 `spring.factories` 文件中查找所有在 `org.springframework.boot.autoconfigure.EnableAutoConfiguration` 项目下的类。在 Spring Boot 框架自带的 `spring-boot-autoconfigure.jar` 中，有几十个不同的配置类，Spring Boot 都会尝试对它们进行加载。但是，由于在配置类和 `@Bean` 定义中存在着各种条件，Spring Boot 无法加载所有这些配置。这种选择性注册 Spring 组件的能力，来自于 Spring Boot 底层的 Spring 框架本身。这些条件范围很广，包括检查某种指定类型的 bean 是否存在，环境配置项是否存在，类路径中是否存在某个类，等等。

我们先来看一下 `CustomerService` 示例。我们要为这个程序构建一个 Spring Boot 版本，使用嵌入式数据库和 Spring Framework `JdbcTemplate`，然后支持构建一个 Web 应用程

序。Spring Boot 会为我们做好所有这一切。回到 `ApplicationConfiguration` 类（参考示例 2-12），我们需要将它改造成一个 Spring Boot 应用程序。

示例2-12 `ApplicationConfiguration.java`类

```
package com.example.boot;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ApplicationConfiguration {
    ❶
}
```

❶ 注意这里没有任何配置！Spring Boot 会为我们配置好所有需要的选项，以及其他事项。

我们新增一个基于 Spring MVC 的控制器，它会暴露一个 REST 端点 `/customers`，响应发送给它的 HTTP GET 请求（如示例 2-13 所示）。

示例2-13 `CustomerRestController.java`类

```
package com.example.boot;

import com.example.Customer;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collection;

    ❶
@RestController
public class CustomerRestController {

    private final CustomerService customerService;

    public CustomerRestController(CustomerService customerService) {
        this.customerService = customerService;
    }

    ❷
    @GetMapping("/customers")
    public Collection<Customer> readAll() {
        return this.customerService.findAll();
    }
}
```

❶ `@RestController` 是另一种原型注解，如 `@Component`。它会告诉 Spring，该组件作为一个 REST 控制器使用。

- ② 我们可以使用 Spring MVC 注释来映射 HTTP 方法，例如在这里，我们可以使用 `@GetMapping` 处理指定端点的 HTTP GET 请求。

可以在 `ApplicationConfiguration` 类中创建一个入口点（如示例 2-14 所示），然后运行该应用程序，并在浏览器中访问 `http://localhost:8080/custom`。从认识上讲，我们更容易理解我们业务逻辑中发生了什么，而且我们做得更少了！

示例2-14 使用Spring MVC构建一个REST终端

```
public static void main(String [] args){
    SpringApplication.run (ApplicationConfiguration.class, args);
}
```

代码变得更少了，但是功能变得更多了。我们知道，在某个地方一定像我们之前一样显式进行了配置。`JdbcTemplate` 是在哪里被创建的呢？实际上，它是在一个名为 `JdbcTemplateAutoConfiguration` 的自动配置类中被创建的，这个类的简略代码如下例 2-15 所示。

示例2-15 JdbcTemplateAutoConfiguration

```
@Configuration ①
@ConditionalOnClass({ DataSource.class, JdbcTemplate.class }) ②
@ConditionalOnSingleCandidate(DataSource.class) ③
@AutoConfigureAfter(DataSourceAutoConfiguration.class) ④
public class JdbcTemplateAutoConfiguration {

    private final DataSource dataSource;

    public JdbcTemplateAutoConfiguration(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    @Primary
    @ConditionalOnMissingBean(JdbcOperations.class) ⑤
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(this.dataSource);
    }

    @Bean
    @Primary
    @ConditionalOnMissingBean(NamedParameterJdbcOperations.class) ⑥
    public NamedParameterJdbcTemplate namedParameterJdbcTemplate() {
        return new NamedParameterJdbcTemplate(this.dataSource);
    }
}
```

- ① 这是一个普通的 `@Configuration` 类。

- ② 只有类型为 `DataSource.class` 和 `JdbcTemplate.class` 的类在路径中时，才会被认为是配置类；否则一定会失败并抛出类似 `ClassNotFoundException` 的错误。
- ③ 如果在应用程序上下文中已经引入了一个 `DataSource bean`，我们希望只使用这个配置类。
- ④ 我们知道，`DataSourceAutoConfiguration` 可以引入一个嵌入式的 H2 数据源，所以这个注解可以保证在 `DataSourceAutoConfiguration` 运行之后启动该配置。如果已经引入了数据库，那么这个配置类将根据数据库进行配置。
- ⑤ 我们希望引入一个 `JdbcTemplate`，但是只有当用户（这里指你和我）还没有在自己的配置类中定义一个同类型 `bean` 的条件下才可以。
- ⑥ 我们希望引入一个 `NamedParameterJdbcTemplate`，但只有当它不存在的时候才可以。

Spring Boot 的自动配置功能大大降低了我们需要实际编写的代码量以及重复工作量，从而我们可以把重点放在业务逻辑上，而由框架来实现细节。如果我们想对系统中某些切面进行细致的控制，只需引入某些其他类型的 `bean`，框架会帮助我们自动替换。Spring Boot 是开闭原则的一个实现：它对扩展开放，但是对修改关闭。

你不需要重新编译 Spring 或 Spring Boot 来满足某些机器的要求。你可能会发现 Spring Boot 应用程序会表现出一些奇怪的、你希望去重新定义的行为。因此，了解如何自定义应用程序以及如何调试十分重要。可以在应用程序启动时指定 `--Ddebug=true` 参数，这样 Spring Boot 会打印出“调试报告”，显示出当前的所有配置项，以及它们是否与期望值匹配。通过调试报告，你可以很容易地检查自动配置类，确定它们的行为是否正确。

Cloud Foundry 平台

Spring Boot 让我们能够专注于应用程序本身，但是我们不想在将应用程序迁移到生产环境之后，就失去了这些新得到的能力。运维应用程序是一项艰巨的但不能忽视的任务。我们的目标是不断将应用程序部署到类似生产的环境中，并且在集成和验收测试过程中进行认证，确保当它被部署到生产环境中后可以正常地运行。重要的是，我们应该尽早并且尽可能频繁地发布应用程序到生产环境中，因为这是唯一一个客户可以验证团队交付产出的地方。

如果应用程序很难被部署到生产环境中，那么开发团队将会不可避免地害怕这个过程，犹豫不决，同时也会增加每次从开发环境部署到生产环境之间的代码量。除此之外，这也增加了尚未部署到生产环境的工作量，这意味着每次部署风险都变得更大，因为每个版本中含有的业务逻辑更多了。为了降低部署到生产环境的风险，我们需要减少每次部

署的代码量，增加部署频率。如果生产环境出现错误，应该尽可能低成本地修复问题，并重新部署修复程序。

这其中的关键，是将整个价值链条中可以自动化的东西全部自动化，从产品管理到生产环境，它们对流程没有增加任何价值。部署不是一个有业务价值的活动，它没有增加流程的价值。因此，你应该将它完全自动化，从而获得更快的速度。

Cloud Foundry 是一个希望帮助你做到这些的云平台。它是一个平台即服务（PaaS）的模式。Cloud Foundry 的关注点，不在基础架构即服务（IaaS）中的硬件、RAM、CPU、Linux 安装和安全补丁上，也不在容器即服务的容器上，而在应用程序及其服务上。运维人员只需要关注应用程序及其服务，不用考虑其他因素。因为我们会在本书后续部分介绍 Cloud Foundry，所以我们这里介绍 Spring 的配置时，先主要介绍如何迭代开发和部署 Spring Boot 应用程序。

Cloud Foundry 有许多实现，都基于开源的 Cloud Foundry 代码。对于本书来说，我们已经将书中所有应用程序都在 Pivotal Web Services (<http://run.pivotal.io>) 上部署及运行过了。Pivotal Web Services 提供了 Pivotal Cloud Foundry (<https://pivotal.io/platform>) 的一部分功能。它被部署在 AWS 的东部区域。如果你想开始使用 Cloud Foundry，PWS 是一个价格实惠的方案，它可以同时服务于大型和小型项目，并且由 Pivotal 的运维团队负责维护。

访问 PWS 的主页面，你可以找到一个登录现有账户，或者注册新账户的地方（如图 2-17 所示）。

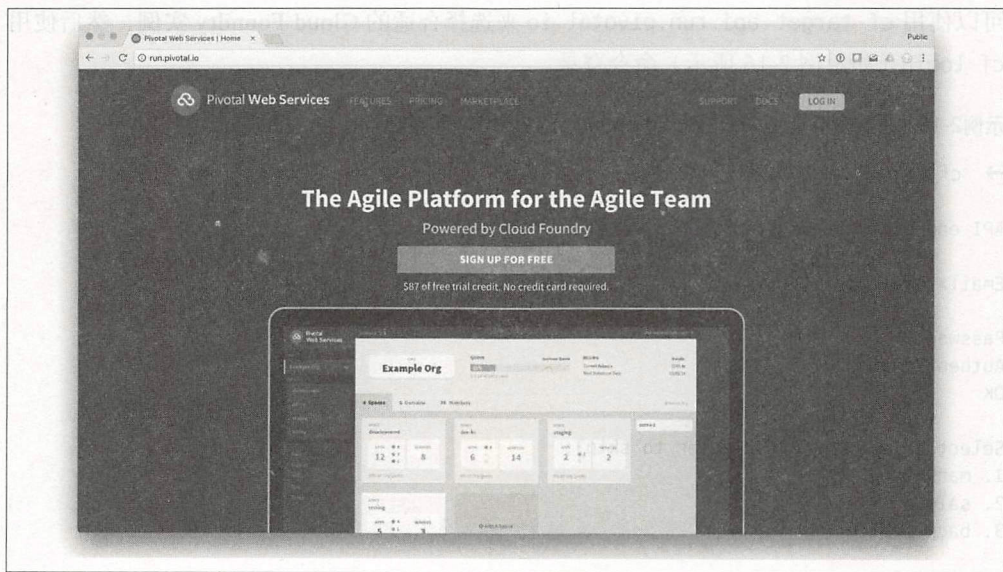


图2-17 PWS主页

登录 PWS 以后，你可以查看账户，并获取已部署应用程序的信息（如图 2-18 所示）。

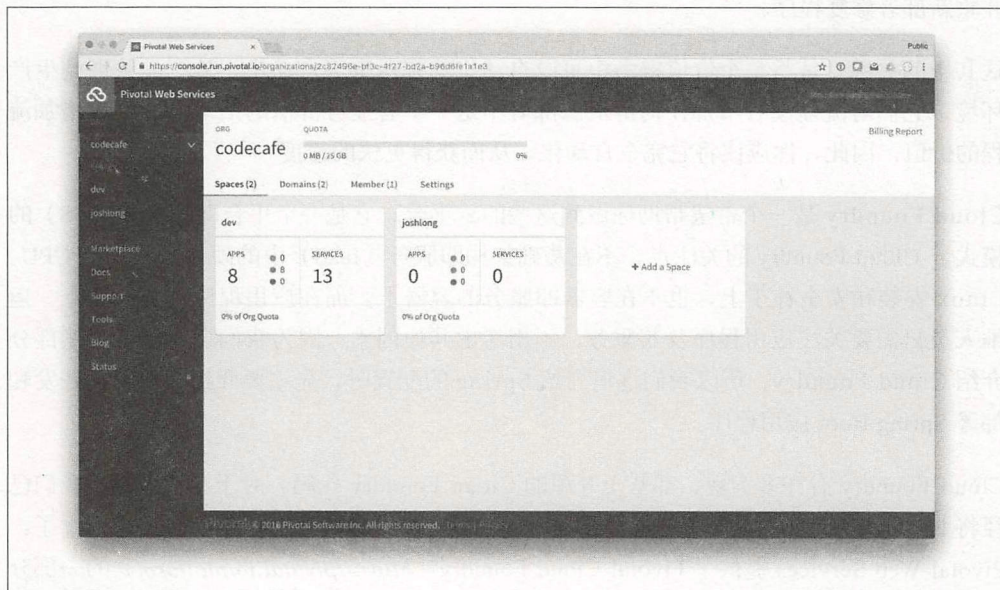


图2-18 PWS控制台

如果你有了一个 PWS 账户，务必先安装 `cf` 命令行接口 (CLI) (<https://docs.cloudfoundry.org/cf-cli/install-go-cli.html>)。在安装之前，需要通过账户来登录网站。

可以使用 `cf target api.run.pivotal.io` 来选择合适的 Cloud Foundry 实例，然后使用 `cf login`（如示例 2-16 所示）命令登录。

示例2-16 进行身份认证并定位Cloud Foundry的某个组织和空间

→ `cf login`

API endpoint: `https://api.run.pivotal.io`

Email> `email@gmail.com`

Password>

Authenticating...

OK

Select an org (or press enter to skip):

1. marketing
2. sales
3. back-office

Org> 1

Targeted org back-office

Targeted space development

API endpoint: <https://api.run.pivotal.io> (API version: 2.65.0)
User: email@gmail.com
Org: back-office ❶
Space: development ❷

❶ 在 Cloud Foundry 中，指定用户可以访问多个组织。

❷ 在该组织内，可能存在着多种环境（例如开发环境、预发布环境和集成环境）。

假设现在我们有一个有效的账户并已经成功登录，我们可以部署 `target/configuration.jar` 这个已有的应用程序。需要使用 `cf create-service` 命令来设置一个数据库（如示例 2-17 所示）。

示例2-17 创建一个MySQL数据库，将其绑定到我们的应用程序，然后启动应用程序

```
cf create-service p-mysql 100mb bootcamp-customers-mysql ❶  
cf push -p target/configuration.jar bootcamp-customers \  
--random-route --no-start ❷  
cf bind-service bootcamp-customers bootcamp-customers-mysql ❸  
cf start bootcamp-customers ❹
```

❶ 首先，需要在 MySQL 服务（名为 `pmysql`）中选择一个 `100mb` 计划的 MySQL 数据库。这里，我们给它分配一个逻辑名称，`bootcamp-customers-mysql`。使用 `cf marketplace` 来查看 Cloud Foundry 服务市场中的其他服务。

❷ 然后将应用程序包 `target/configuration.jar` 推送到 Cloud Foundry 上。创建应用程序和一个随机路由（一个在 `PWS cfapps.io` 域下的 URL），但是我们还不想启动它：它还需要一个数据库才可以启动。

❸ 现在数据库已经存在了，但是我们需要将它绑定到一个应用程序上。最终，绑定操作会暴露应用程序的环境变量，以及环境变量中相关的连接信息。

❹ 现在应用程序已经被绑定到一个数据库上了，我们终于可以启动它了！

当你将应用程序推送到 Cloud Foundry 上时，你其实向它提交了一个应用程序的二进制 `.jar` 文件，而不是一个虚拟机或者 Linux 容器（尽管你可以给它提供一个 Linux 容器）。当 Cloud Foundry 收到 `.jar` 文件时，它将尝试确定应用程序的类型。它是一个 Java 应用程序，一个 Ruby 应用程序，还是一个 `.NET` 应用程序？它最终会判断出是 Java 应用

程序，并将 `.jar` 传递给 Java 的构建包。构建包 *buildpack* 是一个包含了应用程序生命周期中所有需要调用的脚本的目录。你可以通过指定 URL 来覆盖默认的构建包，也可以选择使用默认的构建包。Cloud Foundry 提供了各种不同语言和平台的构建包，包括 Java、.NET、Node.js、Go、Python、Ruby 等。Java 构建包会发现应用程序有一个可执行的 `main (String[] args)` 方法，并且它是独立的。它将拉取最新的 OpenJDK 版本，并指定它作为我们应用程序的运行环境。所有这些配置都被打包成一个 Linux 容器，随后 Cloud Foundry 的调度程序将它部署在一个集群中。Cloud Foundry 可以在一个单独环境中运行数十万个容器。

从这以后，你都可以在 Cloud Foundry 控制台上看到已经部署的应用程序和一个 URL。恭喜你！我们的应用程序现在已经被部署成为了一个 Cloud Foundry 实例。

在我们的应用程序中有一个数据源 bean，当我们将其指向一个 MySQL 数据源，覆盖默认的嵌入式 H2 数据源定义后，Cloud Foundry 会自动将其重新映射到一个独立的、已绑定的 MySQL 服务上。

在我们每次部署应用程序时，可能有很多想要描述的内容。在第一次运行中，我们使用各种 `cf` 命令来配置应用程序，但是很快这些会变得烦琐。相反，我们也可以在 Cloud Foundry 的清单文件（通常名称为 `manifest.yml`）中来配置应用程序。以下是我们目前正常运行的应用程序的一个 `manifest.yml` 文件的内容，其中配置的 MySQL 数据源名称如前文所述（如示例 2-18 所示）。

示例2-18 Cloud Foundry清单

```
---
applications:
- name: bootcamp-customers
  buildpack: https://github.com/cloudfoundry/java-buildpack.git
  instances: 1
  random-route: true
  path: target/spring-configuration.jar
  services:
    - bootcamp-customers-mysql
  env:
    DEBUG: "true"
    SPRING_PROFILES_ACTIVE: cloud
```

- ❶ 为应用程序指定了一个名称。
- ❷ 指定一个构建包。
- ❸ 指定需要使用哪个二进制文件。

- ④ 指定所配置的 Cloud Foundry 服务的依赖。
- ⑤ 指定环境变量来覆盖影响 Spring Boot 的属性。--Ddebug=true (或 DEBUG: true) 会列出自动配置中的所有条件, --Dspring.profiles.active = cloud 指定在 Spring 应用程序中, 应该激活哪个 profile 文件或者哪一组配置。这个配置表示只加载 profile 文件 cloud 中的所有 Spring bean。

现在, 你不再需要运行上面那些 cf 命令了, 只需要运行 `cf push -f manifest.yml` 即可。很快, 你的应用程序就可以启动完毕并正常运行。

到目前为止, 我们已经看到, Cloud Foundry 平台就是通过自动化来获得速度: 平台尽量完成重复的工作, 让你能够专注于核心的业务逻辑。我们按照 Cloud Foundry 独特的方式来工作: 如果你的应用程序需要某些功能, 你可以通过 cf 命令或在 manifest.yml 中尽可能地声明, 随后平台会自动帮你完成。令人惊讶的是, 尽管平台存在很多限制, 但 Cloud Foundry 本身也很容易编程。它提供了一个 rich API, 几乎支持你想要做的一切事情。而且, Spring 和 Cloud Foundry 团队开发了一个 Java 的 Cloud Foundry 客户端, 支持 Cloud Foundry 中的所有主要功能。Cloud Foundry Java 客户端还支持进行更高级别、更细粒度的业务操作, 如同 cf 命令所提供的功能一样。

Cloud Foundry Java 客户端基于 Pivotal Reactor 3.0 项目。反过来, Reactor 又依赖于 Spring Framework 5 中的可响应式 Web 运行时。Cloud Foundry Java 客户端运行速度很快, 因为它基于响应式原则, 几乎完全是非阻塞的。

Reactor 项目反过来又是一个 Reactive Streams 计划的实现。根据网站 (<http://www.reactive-streams.org>) 中介绍的 Reactive Streams 计划, “是为了提供一个非阻塞的异步流处理标准”, 它提供了一种语言和一些 API, 来描述一个潜在的、数据不断异步到达的无限流。使用 Reactor API 可以轻松地编写出可并行运行的代码, 它的目标是避免资源的过度使用, 并有效隔离使用云计算时的阻塞工作。Reactive Streams 计划定义了一个反压力 (backpressure) 的概念, 即订阅者可以通知发布者, 它不希望再收到任何通知。从本质上讲, 它反向推动生产者要限制流量, 直到它有能力处理。

这个 API 的核心是 `org.reactivestreams.Publisher`, 它最终可能会产生零个或多个值。订阅者会订阅来自发布者新数据的通知。Reactor 项目定义了两个常用的发布者规范: Mono 和 Flux。Mono<T> 是一个只产生一个值的 Publisher<T>。而 Flux<T> 是一个产生零个或多个值的 Publisher<T>。

基数

同步

异步

一个

基数

T

Future<T>

Many

Collection<T>

org.reactivestreams.Publisher<T>

我们不深入介绍 Reactive Streams 或者 Reactor 项目，只需要知道它为 Cloud Foundry Java API 出色的效率提供了基础。Cloud Foundry Java API 适用于 cf CLI 命令行非常难以实现的并行化处理。我们已经使用 cf CLI 和 manifest.yml 文件部署了相同的应用程序，现在来看看如何通过 Java 代码做到这一点。在集成测试中使用 Cloud Foundry Java 客户端会非常方便。为了使用 Cloud Foundry Java 客户端，你需要配置一些与 Cloud Foundry 各类子系统能够安全集成的对象，包括日志聚合子系统、Cloud Foundry REST API 以及 Cloud Foundry 身份认证子系统。虽然在这里会演示这些组件的配置（如示例 2-19 所示），但是在 Spring Boot 即将发布的下一个版本中，你可能无须再配置它们。

示例2-19 配置Cloud Foundry Java客户端

```
package com.example;
```

```
import org.cloudfoundry.client.CloudFoundryClient;
import org.cloudfoundry.operations.DefaultCloudFoundryOperations;
import org.cloudfoundry.reactor.ConnectionContext;
import org.cloudfoundry.reactor.DefaultConnectionContext;
import org.cloudfoundry.reactor.TokenProvider;
import org.cloudfoundry.reactor.client.ReactorCloudFoundryClient;
import org.cloudfoundry.reactor.doppler.ReactorDopplerClient;
import org.cloudfoundry.reactor.tokenprovider.PasswordGrantTokenProvider;
import org.cloudfoundry.reactor.uaa.ReactorUaaClient;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
```

```
@SpringBootApplication
```

```
public class CloudFoundryClientExample {
```

```
    public static void main(String[] args) {
        SpringApplication.run(CloudFoundryClientExample.class, args);
    }
```

❶

```

@Bean
ReactorCloudFoundryClient cloudFoundryClient(
    ConnectionContext connectionContext, TokenProvider tokenProvider) {
    return ReactorCloudFoundryClient.builder()
        .connectionContext(connectionContext).tokenProvider(tokenProvider).build();
}

```

②

```

@Bean
ReactorDopplerClient dopplerClient(ConnectionContext connectionContext,
    TokenProvider tokenProvider) {
    return ReactorDopplerClient.builder().connectionContext(connectionContext)
        .tokenProvider(tokenProvider).build();
}

```

③

```

@Bean
ReactorUaaClient uaaClient(ConnectionContext connectionContext,
    TokenProvider tokenProvider) {
    return ReactorUaaClient.builder().connectionContext(connectionContext)
        .tokenProvider(tokenProvider).build();
}

```

④

```

@Bean
DefaultCloudFoundryOperations cloudFoundryOperations(
    CloudFoundryClient cloudFoundryClient, ReactorDopplerClient dopplerClient,
    ReactorUaaClient uaaClient, @Value("${cf.org}") String organization,
    @Value("${cf.space}") String space) {
    return DefaultCloudFoundryOperations.builder()
        .cloudFoundryClient(cloudFoundryClient).dopplerClient(dopplerClient)
        .uaaClient(uaaClient).organization(organization).space(space).build();
}

```

⑤

```

@Bean
DefaultConnectionContext connectionContext(@Value("${cf.api}") String apiHost) {
    if (apiHost.contains("://")) {
        apiHost = apiHost.split("://")[1];
    }
    return DefaultConnectionContext.builder().apiHost(apiHost).build();
}

```

⑥

```

@Bean
PasswordGrantTokenProvider tokenProvider(@Value("${cf.user}") String username,
    @Value("${cf.password}") String password) {
    return PasswordGrantTokenProvider.builder().password(password)
        .username(username).build();
}
}

```

① ReactorCloudFoundryClient 是 Cloud Foundry REST API 的客户端。

- ② `ReactorDopplerClient` 是 Cloud Foundry 基于 websocket 的日志聚合子系统 Doppler 的客户端。
- ③ `ReactorUaaClient` 是 Cloud Foundry 中 UAA 身份认证和授权子系统的客户端。
- ④ `DefaultCloudFoundryOperations` 将一些低级子系统的客户端组合起来，提供一些更粗粒度的操作。从这里开始。
- ⑤ `ConnectionContext` 描述了我们希望定位的 Cloud Foundry 实例。
- ⑥ `PasswordGrantTokenProvider` 描述了我们的身份认证。

要想证明以上概念是很简单的。示例 2-20 是一个简单的示例，它列举出了某个 Cloud Foundry 空间和组织中所有已部署的应用程序。

示例2-20 列出应用程序实例

```
package com.example;

import org.cloudfoundry.operations.CloudFoundryOperations;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
class ApplicationListingCommandLineRunner implements CommandLineRunner {

    private final CloudFoundryOperations cf; ①

    ApplicationListingCommandLineRunner(CloudFoundryOperations cf) {
        this.cf = cf;
    }

    @Override
    public void run(String... args) throws Exception {
        cf.applications().list().subscribe(System.out::println); ②
    }
}
```

① 注入已配置的 `CloudFoundryOperations`。

② 并使用它列举出指定 Cloud Foundry 空间和组织中所有部署的应用程序。

为了演示一个更实际的例子，我们将使用 Java 客户端来部署我们的 `bootcamp-customer` 应用程序。以下是一个简单的集成测试，它配置了一个 MySQL 服务，然后将应用程序推送到 Cloud Foundry（但不启动），绑定环境变量，绑定 MySQL 服务，最后启动应用程序。首先，我们先来看一下如何识别可部署的 `.jar`，以及给应用程序和服务命名的代码（如示例 2-21 所示）。我们将会委托两个组件 `Application Deployer` 和 `ServicesDeployer` 来实现。

示例2-21 配置应用程序集成测试的关键代码

```

package bootcamp;

import org.cloudfoundry.operations.CloudFoundryOperations;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.PropertySource;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.io.File;
import java.time.Duration;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.CyclicBarrier;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = SpringConfigurationIT.Config.class)
public class SpringConfigurationIT {

    @Autowired
    private ApplicationDeployer applicationDeployer;

    @Autowired
    private ServicesDeployer servicesDeployer;

    @Test
    public void deploy() throws Throwable {

        File projectFolder = new File(new File("."), "../spring-configuration");
        File jar = new File(projectFolder, "target/spring-configuration.jar");

        String applicationName = "bootcamp-customers";
        String mysqlSvc = "bootcamp-customers-mysql";

        Map<String, String> env = new HashMap<>();
        env.put("SPRING_PROFILES_ACTIVE", "cloud");

        Duration timeout = Duration.ofMinutes(5);
        servicesDeployer.deployService(applicationName, mysqlSvc, "p-mysql", "100mb")
        ①
        .then(
            applicationDeployer.deployApplication(jar, applicationName, env, timeout,
                mysqlSvc) ②
        ).block(); ③
    }
}

```



```

}

@SpringBootApplication
public static class Config {

    @Bean
    ApplicationDeployer applications(CloudFoundryOperations cf) {
        return new ApplicationDeployer(cf);
    }

    @Bean
    ServicesDeployer services(CloudFoundryOperations cf) {
        return new ServicesDeployer(cf);
    }
}
}

```

- ❶ 首先部署后端服务（一个 MySQL 实例）。
- ❷ 然后部署应用程序，确保五分钟后超时。
- ❸ 然后停止继续测试。

此示例由两个 Publisher 实例组成，一个描述了配置服务所需的步骤，另一个描述了配置应用程序所需的步骤。调用链中最后的 .block() 方法会触发整个语句执行，它是激活整个流程的结尾方法。

ServicesDeployer 可以接受所需的参数并配置我们的 MySQL 实例(如示例 2-22 所示)。如果已经存在一个 MySQL 实例，它也会解除与实例的绑定并删除该实例。

示例2-22 ServicesDeployer

```

package bootcamp;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.cloudfoundry.operations.CloudFoundryOperations;
import org.cloudfoundry.operations.services.CreateServiceInstanceRequest;
import org.cloudfoundry.operations.services.DeleteServiceInstanceRequest;
import org.cloudfoundry.operations.services.ServiceInstanceSummary;
import org.cloudfoundry.operations.services.UnbindServiceInstanceRequest;
import org.reactivestreams.Publisher;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.function.Function;

class ServicesDeployer {

```

```

private final Log log = LoggerFactory.getLog(getClass());

private final CloudFoundryOperations cf;

ServicesDeployer(CloudFoundryOperations cf) {
    this.cf = cf;
}

Mono<Void> deployService(String applicationName, String svcInstanceName,
    String svcTypeName, String planName) {

    return cf.services().listInstances().cache()❶
        .filter(si1 -> si1.getName().equalsIgnoreCase(svcInstanceName))❷
        .transform(unbindAndDelete(applicationName, svcInstanceName))❸
        .thenEmpty(createService(svcInstanceName, svcTypeName, planName));❹
}

private Function<Flux<ServiceInstanceSummary>, Publisher<Void>> unbindAndDelete(
    String applicationName, String svcInstanceName) {
    return siFlux -> Flux.concat(
        unbind(applicationName, svcInstanceName, siFlux),
        delete(svcInstanceName, siFlux));
}

private Flux<Void> unbind(String applicationName, String svcInstanceName,
    Flux<ServiceInstanceSummary> siFlux) {
    return siFlux.filter(si -> si.getApplications().contains(applicationName))
        .flatMap(
            si -> cf.services().unbind(
                UnbindServiceInstanceRequest.builder().applicationName(applicationName)
                    .serviceInstanceName(svcInstanceName).build()));
}

private Flux<Void> delete(String svcInstanceName,
    Flux<ServiceInstanceSummary> siFlux) {
    return siFlux.flatMap(si -> cf.services().deleteInstance(
        DeleteServiceInstanceRequest.builder().name(svcInstanceName).build()));
}

private Mono<Void> createService(String svcInstanceName, String svcTypeName,
    String planName) {
    return cf.services().createInstance(
        CreateServiceInstanceRequest.builder().serviceName(svcTypeName)
            .planName(planName).serviceInstanceName(svcInstanceName).build());
}

```

- ❶ 列举所有的应用程序实例并将它们缓存起来，以便后续的订阅者不必再重新访问 REST 调用。
- ❷ 过滤所有的服务实例，只保留与特定名称匹配的服务实例。

③ 然后解除绑定（如果已经绑定）并删除该服务。

④ 最后重新创建服务。

然后，ApplicationDeployer 会开始配置应用程序（如示例 2-23 所示），它会绑定到我们刚刚使用 ServicesDeployer 配置的服务上。

示例2-23 ApplicationDeployer

```
package bootcamp;

import org.cloudfoundry.operations.CloudFoundryOperations;
import org.cloudfoundry.operations.applications.PushApplicationRequest;
import org.cloudfoundry.operations.applications.
    SetEnvironmentVariableApplicationRequest;
import org.cloudfoundry.operations.applications.StartApplicationRequest;
import org.cloudfoundry.operations.services.BindServiceInstanceRequest;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.io.File;
import java.time.Duration;
import java.util.HashMap;
import java.util.Map;

class ApplicationDeployer {

    private final CloudFoundryOperations cf;

    ApplicationDeployer(CloudFoundryOperations cf) {
        this.cf = cf;
    }

    Mono<Void> deployApplication(File jar, String applicationName,
        Map<String, String> envOg, Duration timeout, String... svcs) {
        return cf.applications().push(pushApp(jar, applicationName))①
            .then(bindServices(applicationName, svcs))②
            .then(setEnvironmentVariables(applicationName, new HashMap<>(envOg)))③
            .then(startApplication(applicationName, timeout));④
    }

    private PushApplicationRequest pushApp(File jar, String applicationName) {
        return PushApplicationRequest.builder().name(applicationName).noStart(true)
            .randomRoute(true)
            .buildpack("https://github.com/cloudfoundry/java-buildpack.git")
            .application(jar.toPath()).instances(1).build();
    }
}
```

```

private Mono<Void> bindServices(String applicationName, String[] svcs) {
    return Flux
        .just(svcs)
        .flatMap(
            svc -> {
                BindServiceInstanceRequest request = BindServiceInstanceRequest.builder()
                    .applicationName(applicationName).serviceName(svc).build();
                return cf.services().bind(request);
            }).then();
}

private Mono<Void> startApplication(String applicationName, Duration timeout) {
    return cf.applications().start(
        StartApplicationRequest.builder().name(applicationName)
            .stagingTimeout(timeout).startupTimeout(timeout).build());
}

private Mono<Void> setEnvironmentVariables(String applicationName,
    Map<String, String> env) {
    return Flux
        .fromIterable(env.entrySet())
        .flatMap(
            kv -> cf.applications().setEnvironmentVariable(
                SetEnvironmentVariableApplicationRequest.builder().name(applicationName)
                    .variableName(kv.getKey()).variableValue(kv.getValue()).build()).then());
}
}

```

- ❶ 首先，我们推送应用程序的二进制包（它的路径），指定一个构建包、内存以及一些部署属性（例如实例个数）。重要的是，我们这时还没有启动应用程序。在绑定环境变量和服务之后，我们才会启动应用程序。如果你使用 Cloud Foundry 清单来进行配置，该步骤将由 Cloud Foundry 自动执行。
- ❷ 将服务实例绑定到我们的应用程序上。
- ❸ 设置应用程序的环境变量。
- ❹ 最后，启动应用程序。

运行整个过程只需要很少的时间，如果你做一些调优，将一些特定流程分支放到其他线程上运行，那么需要的时间会更少。Cloud Foundry 的 Java 客户端非常强大，也是笔者最喜欢的复杂系统描述方法之一。尤其是当你需要部署大量的 shell 脚本时，使用它非常方便。

总结

在本章中，我们简要介绍了 Spring Boot、诸如 Spring Tool Suite 这样的支持工具、如何创建 Java 配置，以及如何将该应用程序迁移到云服务环境。现在我们已经能够自动将代码部署到生产环境了，这样就可以很容易地支持 Jenkins 或 Bamboo 等持续集成环境。在接下来的章节中，我们将更加全面地介绍 Spring Boot 和 Cloud Foundry。

符合十二要素程序风格的配置

在本章中，我们将学习如何将应用程序的配置外部化。

令人迷惑的“配置”合并

我们先来确定一些术语。当我们在 Spring 中讨论配置时，我们通常讨论的是 Spring 框架中各种 `ApplicationContext` 实现 (<http://bit.ly/2rjucwJ>) 的输入形式，它们用来帮助容器理解如何将各种 bean 装配到一起。它可能是一个被包装成 `PathXmlApplicationContext` (<http://bit.ly/2rj2oZb>) 类的 XML 文件，或者是一个被包装成 `AnnotationConfigApplicationContext` (<http://bit.ly/2rjzd8j>) 注解的 Java 类。事实上，当我们谈论后者时，我们一般称其为 Java 配置。

但是，在本章中，我们将了解在十二要素程序宣言 (<http://12factor.net/config>) 中对如何定义配置的描述。在这种情况下，配置是指可能在不同环境下发生变化的字符串，例如密码、端口、主机名或者功能开关等。这种配置避免了将这类值硬编码到代码中。宣言中提供了一个很好的测试配置是否正确的方法，即应用程序的代码库是否可以在任何时候开源，而不会泄露重要的隐私内容？这种配置只是指随着环境不同可能发生变化的值，但是不包括 Spring bean 装配或者 Ruby 路由这样的配置。

Spring 框架对配置的支持

Spring 自从引入 `PropertyPlaceholderConfigurer` (<http://bit.ly/2riQBKw>) 类以来，就已经支持了符合十二要素程序要求的配置。一旦你定义了一个 `PropertyPlaceholderConfigurer` 类的实例，Spring 就会用 `.properties` 文件中解析的值来替换 XML 配置中的字符串。Spring 从 2003 年开始提供 `PropertyPlaceholderConfigurer` ([http:// bit.ly/2rj5ixb](http://bit.ly/2rj5ixb)) 类。

Spring 2.5 引入了对 XML 名称空间的支持，以及支持使用属性占位符的 XML 名称空间。这使我们可以将 XML 配置中的 bean 定义字面值，替换为外部属性文件（在本例中为 `simple.properties`，它可能位于应用程序的类路径上，或者应用程序之外）中的值。

十二要素风格的配置旨在消除魔法字符串——在程序代码中硬编码的数据库地址、密码、端口等值。如果这些配置是外部化的，那么不需要重新编译代码就可以更换配置。

PropertyPlaceholderConfigurer

我们来看一个使用 `PropertyPlaceholderConfigurer`、Spring XML bean 定义和一个外部 `.properties` 文件的例子。我们只是想打印出属性文件中的值，如示例 3-1 所示。

示例3-1 属性文件: `some.properties`

```
configuration.projectName = Spring Framework
```

这是一个 Spring `ClassPathXmlApplicationContext` 类，所以我们使用 Spring 的 context XML 名称空间并指向 `some.properties` 文件。然后，在 bean 定义中，使用类似 `${configuration.projectName}` 的字符串，Spring 就可以在运行时使用属性文件（如示例 3-2 所示）中的值来替换它们了。

示例3-2 Spring XML 配置文件

```
<context:property-placeholder location="classpath:some.properties"/> ❶

<bean class="classic.Application">
  <property name="configurationProjectName" value="${configuration.projectName}"/>
</bean>
```

❶ `classpath`：该位置是指当前已编译代码包（`.jar`、`.war` 等）中的某个文件。Spring 支持许多替代方法，包括 `file`：和 `url`：，这使得配置文件可以存在于代码包之外。

最后，通过一个 Java 类来把它们结合到一起（如示例 3-3 所示）。

示例3-3 一个使用属性值进行配置的Java类

```
package classic;

import org.apache.commons.logging.LogFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {

  public static void main(String[] args) {
```

```

    new ClassPathXmlApplicationContext("classic.xml");
}

public void setConfigurationProjectName(String pn) {
    LoggerFactory.getLogger(getClass()).info("the configuration project name is " + pn);
}
}
}

```

第一个例子使用了 Spring 的 XML bean 配置。对于使用 Java 配置的开发人员来说，Spring 3.0 和 3.1 已经改进了很多，引入了 `@Value` 注解和 `Environment` 接口。

Environment 接口和 @Value 注解

`Environment` (<http://bit.ly/2s3GiHf>) 接口提供了运行中应用程序及其运行时环境之间的隔离，并允许应用程序提出关于环境的问题（如“当前平台的 `line.separator` 配置是什么？”）。`Environment` 就好像一个键值对的地图一样。你可以在 `Environment` 中指定一个 `Property Source`，来配置读取这些值的位置。默认情况下，Spring 会加载系统环境中的键值对，例如 `line.separator`。还可以使用 `@PropertySource` 注解，以类似于之前使用 Spring 的属性占位符的方式，告诉 Spring 从指定文件中来加载配置项。

通过 `@Value` 注解，可以将环境中的值注入构造函数、setter 方法、字段等中。如果像我们在示例 3-4 中一样，注册了一个 `PropertySourcesPlaceholderConfigurer` 类 (<http://bit.ly/2s3TQCz>)，那么就可以使用 Spring 表达式语言或者属性占位符语法来计算这些值了。

示例3-4 注册一个 `PropertySourcesPlaceholderConfigurer`

```

package env;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.Environment;

```

```

import javax.annotation.PostConstruct;

```

```

❶
@Configuration
@PropertySource("some.properties")
public class Application {

```



```

private final Log log = LoggerFactory.getLog(getClass());

public static void main(String[] args) throws Throwable {
    new AnnotationConfigApplicationContext(Application.class);
}

2
@Bean
static PropertySourcesPlaceholderConfigurer pspc() {
    return new PropertySourcesPlaceholderConfigurer();
}

3
@Value("${configuration.projectName}")
private String fieldValue;

4
@Autowired
Application(@Value("${configuration.projectName}") String pn) {
    log.info("Application constructor: " + pn);
}

5
@Value("${configuration.projectName}")
void setProjectName(String projectName) {
    log.info("setProjectName: " + projectName);
}

6
@Autowired
void setEnvironment(Environment env) {
    log.info("setEnvironment: " + env.getProperty("configuration.projectName"));
}

7
@Bean
InitializingBean both(Environment env,
    @Value("${configuration.projectName}") String projectName) {
    return () -> {
        log.info("@Bean with both dependencies (projectName): " + projectName);
        log.info("@Bean with both dependencies (env): "
            + env.getProperty("configuration.projectName"));
    };
}

@PostConstruct
void afterPropertiesSet() throws Throwable {
    log.info("fieldValue: " + this.fieldValue);
}
}

```

- ❶ @PropertySource 注解就像是一个“属性占位符”的快捷方式,通过一个 .properties 文件来配置 PropertySource。
- ❷ 需要将 PropertySourcesPlaceholderConfigurer 注册成一个“静态 (static)” bean, 因为它是 BeanFactoryPostProcessor 的一个实现, 并且必须在 Spring bean 初始化生命周期的早期阶段进行调用。当你使用 Spring 的 XML bean 配置时, 可以忽略这个差异。
- ❸ 可以使用 @Value 注解来修饰字段 (但是不要这样做! 它会使测试失败)。
- ❹ 或者你可以使用 @Value 注解来修饰构造函数参数。
- ❺ 也可以使用 setter 方法。
- ❻ 也可以注入 Spring Environment 对象并手动解析配置项。
- ❼ 也可以在 Spring @Bean 的配置方法参数中, 使用 @Value 注解来标注方法参数。

以上示例程序从文件 simple.properties 中加载配置项, 然后通过不同的方式来读取 configuration.projectName 的值。

Profile

Environment 接口还引入了 *Profile* (<http://bit.ly/2s3RjbM>) 的概念。它可以通过一些标签 (Profile) 对 bean 进行分类。Profile 可以用来描述从一个环境变化到另一个环境的 bean 和 bean 图。可以一次激活一个或多个 profile。没有被分配某个 profile 的 bean 总是会被激活。只有当没有其他活动的 profile 时, 才会激活 *default* 配置文件中的 bean。可以在 XML 的 bean 定义中指定 profile 属性, 或者在标签类、配置类、单个 bean 或 @Bean 方法上使用 @Profile 注解。

Profile 允许你描述需要在不同环境中创建的一组 bean。例如, 你可能会在本地 dev profile 中使用嵌入式的 H2 javax.sql.DataSource, 然后当 prod profile 被激活时, 切换到一个通过 JNDI 查询或者通过读取 Cloud Foundry (<http://cloudfoundry.org>) 中环境变量获得的 PostgreSQL javax.sql.DataSource。在这两种情况下, 你的代码都可以正常工作, 都会获得一个 javax.sql.DataSource, 但是具体使用哪个实例, 是由当前活动的 profile 决定的 (如示例 3-5 所示)。

示例3-5 该示例演示了@Configuration类可以加载不同的profile，并根据激活的profile提供不同的bean package profiles;

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.*;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.core.env.Environment;
import org.springframework.util.StringUtils;
```

```
@Configuration
public class Application {
```

```
    private Log log = LogFactory.getLog(getClass());
```

```
    @Bean
    static PropertySourcesPlaceholderConfigurer pspc() {
        return new PropertySourcesPlaceholderConfigurer();
    }
}
```

①

```
@Configuration
@Profile("prod")
@PropertySource("some-prod.properties")
public static class ProdConfiguration {
```

```
    @Bean
    InitializingBean init() {
        return () -> LogFactory.getLog(getClass()).info("prod InitializingBean");
    }
}
```

```
@Configuration
@Profile({ "default", "dev" })
```

②

```
@PropertySource("some.properties")
public static class DefaultConfiguration {
```

```
    @Bean
    InitializingBean init() {
        return () -> LogFactory.getLog(getClass()).info("default InitializingBean");
    }
}
```

③

```
@Bean
InitializingBean which(Environment e,
                        @Value("${configuration.projectName}") String projectName) {
    return () -> {
        log.info("activeProfiles: '"
            + StringUtils.arrayToCommaDelimitedString(e.getActiveProfiles()) + "'");
        log.info("configuration.projectName: " + projectName);
    };
}
```

```

};
}

public static void main(String[] args) {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext();
    ac.getEnvironment().setActiveProfiles("dev"); ❹
    ac.register(Application.class);
    ac.refresh();
}
}

```

- ❶ 只有当 prod profile 处于激活状态时，才会使用这里的配置类和 @Bean 定义。
- ❷ 这里的配置类和 @Bean 定义，只有在 dev profile 被激活，或者没有任何 profile（包括 dev）被激活时才会使用。
- ❸ 这里的 InitializingBean 只是记录当前激活的 profile，并注入最终由属性文件决定的值。
- ❹ 我们可以很容易用编程的方式，激活某个（或多个）profile。

除了使用配置项 `spring_profiles_active` 或者 `spring.profiles.active` 来激活 profile 外，你还可以使用环境变量（例如 `SPRING_PROFILES_ACTIVE`）、JVM 属性（`-Dspring.profiles.active=..`）、Servlet 应用程序初始化参数，或者以编程的方式设置当前活动的 profile。

启动配置

Spring Boot (<http://spring.io/projects/spring-boot>) 大大改进了启动的过程。Spring Boot 默认会自动从几个固定的位置来加载属性。命令行参数会覆盖由 JNDI 提供的属性值，而 JNDI 中的属性值又会覆盖由 `System.getProperties()` 提供的属性值，依次类推。

- 命令行参数
- 从 `java:comp/env` 获取到的 JNDI 属性
- `System.getProperties()` 的属性
- 操作系统的环境变量
- 文件系统上的外部属性文件：(config /)? application.(yml.properties)
- 归档 (config /)? application.(yml.properties) 文件中的属性文件
- 配置类上的 @PropertySource 注解
- `SpringApplication.getDefaultProperties()` 提供的默认属性

如果某个 profile 处于激活状态 (<http://bit.ly/2s3ytl0>)，它也将根据 profile 的名称自动读

取配置文件，例如 `src/main/resources/application-foo.properties`，其中 `foo` 是当前的 `profile`。

如果 SnakeYAML 库 (<https://bitbucket.org/asomov/snakeyaml>) 在类路径中，那么它也将遵循基本相同的约定，自动加载 YAML 文件。



在 YAML 规范的页面中写道 (<http://yaml.org>)，“YAML 是为所有编程语言提供的，对人类友好的数据序列化标准”。YAML 用一个层级的方式来表示值。相比传统的 `.properties` 文件用点 (“.”) 来表示层级结构，YAML 文件使用换行符和额外缩进来表示层级结构。如果你有大量层级化的配置项，YAML 可以避免指定通用的根节点。

示例 3-6 是 `.yaml` 文件的一个例子。

示例3-6 一个 `application.yml` 属性文件，其中数据是分层级的

```
configuration:
  projectName : Spring Boot
management:
  security:
    enabled: false
```

在通常情况下，Spring Boot 也可以更容易获得正确的结果。它将 `-D` 参数作为属性提供给 `java` 进程和环境变量。它甚至规定了一些规范，所以环境变量 `$ CONFIGURATION_PROJECTNAME` 或者形式为 `-Dconfiguration.projectname` 的 `-D` 参数，都可以通过键 `configuration.projectName` 来访问，就像之前提到的 `spring_profiles_active` 一样。

因为配置值是字符串，如果配置值太多，那么很难保证这些配置不会被硬编码在代码中。为了解决这个问题，Spring Boot 引入了 `@ConfigurationProperties` 组件类型。当你使用 `@ConfigurationProperties` 注解来标注一个 POJO 类（一个普通的、旧的 Java 对象）并指定一个前缀时，Spring 将尝试将以该前缀开头的所有属性，映射到 POJO 的属性上。在下面的示例中，`configuration.projectName` 的值将被映射到 POJO 类的一个实例上，然后所有代码都可以读取到这些（类型安全的）值。在这种方式下，你只需要在一个地方对一个（字符串）键进行映射即可（如示例 3-7 所示）。

示例3-7 自动从 `src/main/resources/application.yml` 文件解析属性

```
package boot;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.stereotype.Component;
```

❶

```
@EnableConfigurationProperties
```

```
@SpringBootApplication
```

```
public class Application {
```

```
    private final Log log = LoggerFactory.getLog(getClass());
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(Application.class);
```

```
    }
```

```
    @Autowired
```

```
    public Application(ConfigurationProjectProperties cp) {
```

```
        log.info("configurationProjectProperties.projectName = "
```

```
            + cp.getProjectName());
```

```
    }
```

```
}
```

❷

```
@Component
```

```
@ConfigurationProperties("configuration")
```

```
class ConfigurationProjectProperties {
```

```
    private String projectName; ❸
```

```
    public String getProjectName() {
```

```
        return projectName;
```

```
    }
```

```
    public void setProjectName(String projectName) {
```

```
        this.projectName = projectName;
```

```
    }
```

```
}
```

❶ @EnableConfigurationProperties 注解会告诉 Spring 将属性映射到标记 @ConfigurationProperties 注解的 POJO 实例上。

❷ @ConfigurationProperties 会告诉 Spring，这个 bean 将被作为所有以 configuration 开头的属性的根，configuration 后面的内容会被映射到对象的属性上。

❸ projectName 字段的值，最终会与配置 configuration.projectName 的值一样。

Spring Boot 广泛使用了 @ConfigurationProperties 机制，让用户能够对系统进行配置。你也可以查看在系统中可以更改哪些属性，例如，给基于 Spring Boot 的 Web 应用程序添加一个 org.springframework.boot:spring-boot-starter-actuator 依赖项，然后访

访问 <http://127.0.0.1:8080/configprops>。



第 13 章会进一步讨论 Actuator 端点。这个端点默认是不能被访问的，需要指定用户名和密码。你可以在 `application.properties` 文件（或 `application.yml`）中指定 `management.security.enabled=false`，来禁用安全保护（如果只是想看一眼而已）。

这样会根据运行时类路径中显示的不同类型，列出所支持的配置属性清单。当添加更多的 Spring Boot 类型时，你会看到更多的属性。这个端点也会显示 `@ConfigurationProperties` 注解所暴露出的 POJO 属性。

使用 Spring Cloud Config Server 进行中心化、日志型的配置

到目前为止一切都还顺利，但我们需要做得更好。我们仍然无法满足一些常见的应用场景：

- 更改应用程序的配置，需要重新启动。
- 没有可追溯性。我们如何确定在生产环境中执行了哪些变更，以及如何在必要时回滚它们？
- 配置是分散的。无法立刻找到需要变更的配置项。
- 在安全方面不支持加密和解密。

Spring Cloud Config Server

可以通过将配置存储在单个目录中，并让所有应用程序指向该目录的方式，将配置中心化。也可以使用 Git 或 Subversion 对目录进行版本控制，提供审计和日志的支持。但是这仍不能解决后面两个需求，我们还需要使用一些更复杂的手段。接下来，我们探讨 Spring Cloud Config Server (<http://cloud.spring.io/spring-cloud-config/>)。Spring Cloud 提供了一个配置服务器及其客户端。

Spring Cloud Config Server 是一个 REST API，我们的客户端需要连接它来进行配置。它也是通过版本控制来管理一个配置仓库的。它位于我们的客户端和配置仓库之间，其可以安全地处理从客户端到服务的通信，以及从服务到配置仓库的通信。Spring Cloud Config 客户端为客户端应用程序提供了一个新的作用域 `refresh`，它允许我们不需重启应用程序就可以重新配置 Spring 组件。



像 Spring Cloud Config Server 这样的技术非常重要,但是它们会增加运维开销。理想情况下,应该由平台来自动化地管理这些工具。如果你使用的是 Cloud Foundry,则在服务分类中会有一个基于 Spring Cloud Config Server 的 Config Server 服务。

我们来看一个简单的例子。首先,搭建一个 Spring Cloud Config Server。许多 Spring Boot 应用程序可能会访问单独的配置服务。你需要让它在某个地方运行一次。然后,所有其他的服务只需要知道在哪里可以找到配置服务。配置服务充当了一种从在线或本地 Git 仓库中,读取配置项键值的代理。你只需要在 Spring Boot 应用程序中添加 `org.springframework.cloud:spring-cloud-config-server` 就可以引入 Spring Cloud Config Server (如示例 3-8 所示)。

示例3-8 使用 `@EnableConfigServer` 注解来构建一个配置服务器

```
package demo;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;
```

❶

```
@SpringBootApplication
@EnableConfigServer
public class Application {
```

```
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

❶ `@EnableConfigServer` 会安装 Spring Cloud Config Server。

示例 3-9 展示了对配置服务器的配置。

示例3-9 配置服务器的配置,包含在 `src / main / resources / application.yml` 文件中

```
server.port=8888
spring.cloud.config.server.git.uri=\
    https://github.com/cloud-native-java/config-server-configuration-repository ❶
```

❶ Spring Cloud Config Server 只有通过本地或网络 (例如, GitHub, <http://github.com>) 的方式指向当前工作的 Git 仓库,才可以使用。

在这个例子中,展示了如何让 Spring Cloud 配置服务,在 GitHub 的 Git 仓库中查找配置文件。虽然我们指定了一个 GitHub 仓库,但是任何有效的 Git URI 都可以。事实上,它甚至不必是 Git,你可以使用 Subversion 甚至随便一个目录 (虽然强烈不建议这么做)。

虽然在示例中我们硬编码了仓库的 URI 地址，但是你也可以指定一个 `-D` 参数、`--` 参数或环境变量。

Spring Cloud Config 客户端

为了适应 Spring Cloud 的体系，应该为 Spring Boot 应用程序提供一个名称。可以通过 `spring.application.name` 来设置一个独特的、有用的、易于记忆的名字，就像 `customer-service` 一样。基于 Spring Cloud 的服务会默认去寻找一个 `src / main / resources/bootstrap`. (properties 或者 yml) 文件，以便来启动服务。在那个文件中，Spring Cloud 会查找服务的名称 (`spring.application.name`) 和 Spring Cloud Config Server 的位置，来获取应用程序的配置。该文件 (`bootstrap.properties`) 会比其他属性文件 (包括 `application.yml` 或者 `application.properties`) 更早加载。这是因为它需要告诉 Spring，去哪里找到应用程序的其余配置项。如果你有一个 `application.properties` 和一个 `bootstrap.properties`，`bootstrap.properties` 会先被加载。

Config Server 会管理一个全部都是 `.properties` 或 `.yml` 文件的目录。Spring Cloud 通过将客户端的 `spring.application.name` 与目录中的配置文件进行匹配，来为已连接的客户端提供配置。因此，自身名称为 `foo-service` 的配置客户端，它对应的配置文件为 `foo-service.yml` 或者 `foo-service.properties`。

你可以运行 Config Server 并验证配置服务是否正常工作。可以在浏览器中访问 `http://localhost:8888/SERVICE/master`，其中 `SERVICE` 是你在 `spring.application.name` 中指定的名称。在我们的 Git 仓库中，有一个名为 `configuration-client.properties` 的文件，因此你可以使用 `configuration-client` 作为 `SERVICE` 的值。我们可以在图 3-1 中看到到这个配置文件创建的 JSON。



图3-1 Spring Cloud Config Server的输出，确认了它可以看到我们Git仓库中的配置

示例 3-10 展示了 `bootstrap.yml` 中的配置客户端。

示例3-10 bootstrap.yml示例

```
spring:
  application:
    name: configuration-client
  cloud:
    config:
      uri: ${vcap.services.configuration-service.credentials.uri:http://localhost:8888}
```

Spring Cloud Config Server 通过匹配 `spring.application.name` 名称，返回客户端指定的配置，但它还可以从 Spring Cloud Config Server 仓库中的 `application.properties` 或者 `application.yml` 文件，返回对每个客户端可见的全局配置。

配置服务会返回 `application. (properties 或 yml)` 文件中的所有配置值，以及 `configuration-client. (yml 或 properties)` 中所有服务指定的配置。它还可以加载指定服务和 `profile` 中的配置，例如 `configuration-client-dev.properties`，其中 `configuration-client` 是客户端的名称，`dev` 是客户端正在运行的 `profile` 的名称。

从任何 Spring Boot 应用程序的角度来看，Spring Cloud Config Server 中的值只是另一个 `PropertySource` 对象，可以通过所有正常的方式进行解析，即通过标注 `@Valid` 注解的类成员，或者通过 `Environment` 接口。

安全

如果你的 Git 仓库是受到保护的（例如，使用 HTTP BASIC），则需要为 Spring Cloud Config Server 定义 `spring.cloud.config.server.git.username` 和 `spring.cloud.config.server.git.password` 两个属性，以允许它访问受保护的 Git 仓库。

你也可以使用 HTTP BASIC 身份认证来保护 Spring Cloud Config Server 本身。最简单的就是增加 `org.springframework.boot:spring-boot-starter-security` 依赖，然后定义一个 `security.user.name` 和一个 `security.user.password` 属性。因为 `spring-boot-starter-security` 启动器会引入 Spring Security，所以你可以插入指定的 Spring Security `UserDetailsService` 实现，来覆盖身份认证默认的处理方式。

Spring Cloud Config 客户端可以将 `spring.cloud.config.uri` 值中的用户名和密码进行编码，例如 `https:// user:secret@host.com`。

可刷新的配置

中心化配置是一个强大的功能，但对配置的更改不会立即对依赖于它的 `bean` 起作用。

Spring Cloud 的 *refresh* 作用域（以及便捷的 `@RefreshScope` 注解）提供了一个解决方案。我们先来看一下示例 3-11 中的 `ProjectNameRestController`。

示例3-11 一个Config Server客户端应用程序

```
package demo;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

❶

```
@RestController
```

```
@RefreshScope
```

```
class ProjectNameRestController {
```

```
    private final String projectName;
```

```
    @Autowired
```

```
    public ProjectNameRestController(
```

```
        @Value("${configuration.projectName}") String pn) { ❷
```

```
        this.projectName = pn;
```

```
    }
```

```
    @RequestMapping("/project-name")
```

```
    String projectName() {
```

```
        return this.projectName;
```

```
    }
```

```
}
```

❶ `@RefreshScope` 注解使这个 bean 可刷新。

❷ Spring 会把 Config Server 中的配置值，作为 Environment 中的另一个 PropertySource 来解析。

`ProjectNameRestController` 被标注了 `@RefreshScope` (<http://bit.ly/2s40VDm>) 注解，这个注解是 Spring Cloud 提供的一个作用域，它允许任何 bean 重新创建自己（并从配置服务重新读取配置值）。在这种情况下，只要触发了 *refresh* 事件，就会重新创建 `ProjectNameRestController`，初始化其生命周期，并重新建立 `@Value` 和 `@Autowired` 注入。

基本上，所有 *refresh* 范围的 bean 都会在收到一个类型为 `RefreshScopeRefreshedEvent` 的 Spring `ApplicationContext` 事件时，自动刷新自己。默认情况下，Spring 会重新创建所有标记了 `@RefreshScope` 注解的 bean，而且是直接丢弃整个 bean 并创建一个新的 bean。如果组件的可刷新状态没有被绑定到 Spring Cloud Config 服务器中的其他值上，那么都应该能够响应这个事件。示例 3-12 中展示了一个简单的组件，当每次刷新事件时

计数器加 1。

示例3-12 一个Config Server客户端应用程序

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.cloud.context.scope.refresh.RefreshScopeRefreshedEvent;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

import java.util.concurrent.atomic.AtomicLong;

@Component
public class RefreshCounter {

    private final Log log = LogFactory.getLog(getClass());

    private final AtomicLong counter = new AtomicLong(0); ❶

    ❷
    @EventListener
    public void refresh(RefreshScopeRefreshedEvent e) {
        this.log.info("The refresh count is now at: "
            + this.counter.incrementAndGet());
    }
}
```

❶ 该组件表示一个原子计数器。

❷ 每次检测到 RefreshScopeRefreshedEvent 事件时都会自动更新。

我们有很多种方法来触发它。可以通过向 <http://127.0.0.1:8080/refresh>（一个自动对外暴露的 Spring Boot Actuator 端点）发送一个空的 POST 请求来触发刷新。示例 3-13 显示了如何使用 curl 命令来完成。

示例3-13 使用Spring Cloud Config Client调用单个实例的/refresh Actuator端点，来触发刷新事件

```
curl -d{} http://127.0.0.1:8080/refresh
```

除此之外，还可以使用自动配置的 Spring Boot Actuator JMX refresh 端点，如图 3-2 所示。

要想看到效果，你可以更改 Git 中的配置文件，然后通过 `git commit` 命令提交它们。接下来，可以调用应用配置变更的节点（注意不是配置服务器）上的 REST 端点或 JMX 端点。

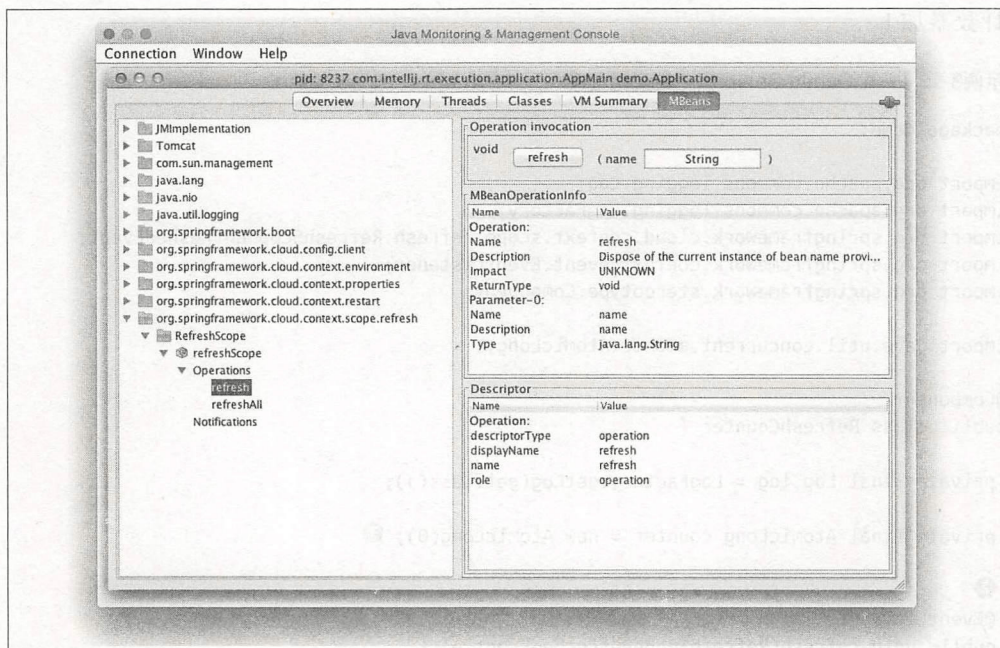


图3-2 使用jconsole激活refresh（或refreshAll）Actuator端点

这两种 Spring Boot Actuator 端点都只支持单个 `ApplicationContext` 实例。如果你有 10 个正在运行的应用程序，并且想要同时查看所有 10 个更新的配置，那么需要在每个实例上调用 Actuator。这种方式的可扩展性很差！

Spring Cloud Bus 支持一次刷新多个 `ApplicationContext` 实例（例如，多个节点）。Spring Cloud Bus (<http://cloud.spring.io/spring-cloud-bus/>) 通过 Spring Cloud Stream 总线来连接所有的服务。Spring Cloud Stream 通过 binder 抽象，支持 RabbitMQ、Apache Kafka、Reactor Project 等多种消息通知技术。我们将在第 10 章中了解有关 Spring Cloud Stream 的更多信息。

这个功能非常强大。你可以通过向消息总线发送一个消息，来通知一个（或几千个）微服务刷新自己。这样不仅可以防止停机，而且比一个一个重启服务或节点效率更高。这里将 RabbitMQ (`org.springframework.cloud:spring-cloud-starter-bus-amqp`) 的 Spring Cloud Bus 依赖添加到类路径中。

Spring Boot 的 RabbitMQ 自动配置，将默认尝试连接到本地的 RabbitMQ 实例。可以使用 Spring Environment 中提供的属性来配置指定的主机和端口。可以将这些配置都保存在 Spring Cloud Config Server 中，这样使用起来更方便。这样，连接到配置服务的所有服务都可以访问正确的 RabbitMQ 实例（如示例 3-14 所示）。

示例3-14 指定一个RabbitMQ ConnectionFactory

```
spring:
  rabbitmq:
    host: my-rmq-host
    port: 5672
    username: user
    password: secret
```

Spring Boot AMQP 自动配置会读取这些配置值，并创建一个 `ConnectionFactory` 实例。Spring Cloud Bus 客户端负责监听消息，当收到消息时会触发一个刷新事件。如果在 Spring 应用程序上下文中有多个 `ConnectionFactory` 实例，你可以通过 `@BusConnectionFactory` 来限定具体使用哪个实例。对于其他用于常规性、非总线相关处理的实例，可以使用 Spring 限定符注解 `@Primary` 来进行标注。

Spring Cloud Bus 开放了另一个 Actuator 端点，`/bus/refresh`，它会向连接的 RabbitMQ 代理发布一个消息，触发所有连接的节点刷新自己。可以使用 `spring-cloud-starter-bus-amqp` 自动配置将以下消息发送给任意节点，它会在所有连接的节点上触发刷新（如示例 3-15 所示）。

示例3-15 使用Spring Cloud Event Bus触发刷新事件

```
curl -d{} http://127.0.0.1:8080/bus/refresh
```

总结

这一章介绍了很多功能！有了这些功能，你可以很容易地构建一个程序包，而且不需要对它做任何改动，就可以将它从一个环境迁移到另一个环境。



一个云原生的应用程序，为了能够达到快速响应，我们会从组件到整个系统的各个层面上对其进行优化，而测试就是推动这个迭代的主要方式。随着微服务的到来，Spring Boot 提供了强大的集成测试支持：支持系统中各组件之间的集成。

随着应用程序变得越来越分布式，我们如何编写有效的测试用例发生了很大变化。集成测试的实践侧重于针对一组相互依赖的软件模块，编写和执行测试用例。集成测试是软件开发中的一种标准实践，编写独立模块或组件的开发人员，尤其是在对影响整体集成的代码进行更改时，能够自动执行一组已经存在的测试用例，从而确保预期的集成功能正常工作。集成测试往往需要在共享的集成环境中执行测试用例。在这种情况下，应用程序可能需要同时共享外部资源，例如数据库或应用程序服务器。

云原生应用程序的设计初衷在于利用云环境的临时性，我们应该专注于如何设计集成测试，使它们能够在与其他应用程序无关的临时环境中执行。在许多情况下，云原生应用程序依赖于一些后端服务。必要的时候，你应该模拟这些依赖的服务，以便可以在一个独立的构建和测试环境中执行集成测试。如果无法模拟外部的依赖服务，那么应该根据实际情况来配置依赖的后端服务，并在测试完成后关闭它们。

在本章中，我们将重点讨论与测试有关的内容，并且集中在两个主要的集成测试话题上。第一个话题是关于如何为 Spring Boot 应用程序设计和创建基本的集成测试用例的。在这里将介绍一些 Spring 提供的工具和功能，它们对于我们创建与外部依赖解耦的集成测试非常重要。

第二个话题是关于如何在微服务体系结构中进行端到端的集成测试的。我们将重点关注如何在一个模拟生产环境的测试环境中，针对一系列不同服务执行各种类型的测试。



测试的构成

本章我们假定你对 JUnit 和 Maven 已经足够了解，所以特别关注集成测试。尽管很多企业会为集成测试建立一个单独的测试工程，但是在我们的例子中，所有的测试代码都会存放在 `src/test/{java, resources}` 目录中，这样我们可以不断地、快速运行这些测试用例，并且降低运行的频率，或许只有在每次提交代码或者进行了重大重构之后才运行它们。无论你的企业采用何种方式，一定确保向生产环境部署二进制包之前，能够自动地运行所有测试用例。

在 Spring Boot 中进行测试

Spring Boot 应用程序的测试分为两种独立的测试类型：单元测试和集成测试。集成测试是指在测试执行期间需要访问 Spring 的应用程序上下文（`ApplicationContext`）。而单元测试不需要访问 Spring 应用程序上下文。

你可以在 `pom.xml` 中添加 test 作用域依赖项 `org.springframework.boot:spring-boot-starter-test`，来启用 Spring Boot 对测试的支持。如果你使用 Spring Initializr (<http://start.spring.io>) 来生成新的项目（你应该这样做），这些都已经为你实现了。

在示例 4-1 中，在 `ApplicationTests` 类中包含了一个简单的集成测试。这个名为 `contextLoads` 的测试，目的是为了确认 `ApplicationContext` 已经被成功地加载，并注入名为 `applicationContext` 的字段中。Spring 中的集成测试就像“hello world”一样简单。

示例 4-1 对 `ApplicationContext` 加载的基本集成测试

```
package demo;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.junit4.SpringRunner;

@SpringBootTest
@RunWith(SpringRunner.class)
public class ApplicationTests {

    @Autowired
    private ApplicationContext applicationContext;

    @Test
    public void contextLoads() throws Throwable {
```




```

    Assert.assertNotNull("the application context should have loaded.",
        this.applicationContext);
}
}

```

`ApplicationTests` 类中有两个注解，分别名为 `@RunWith` 和 `@SpringBootTest`。`@RunWith` 注解只能在 JUnit 框架中使用。`@RunWith` 注解告诉 JUnit 使用哪个测试运行器策略。在这种情况下，我们想用 *Spring TestContext* 框架来运行我们的测试，它是 Spring 框架中的一个模块，为 Spring 应用程序提供了通用的测试支持。



`SpringRunner` 是在 Spring Boot 1.4 中引入的。相较于你可能更熟悉的 `SpringJUnit4ClassRunner` 类，`SpringRunner` 更加好用并且可以支持 JUnit 5。

`ApplicationTests` 类的第二个注解是 `@SpringBootTest`。`@SpringBootTest` 注解表示这个类是一个 Spring Boot 测试类，支持通过扫描 `ContextConfiguration` 来加载 `ApplicationContext`。如果没有为 `@SpringBootTest` 注解指定 `ContextConfiguration` 参数，默认会通过扫描源代码根目录中类的 `@SpringBootConfiguration` 注解来加载 `ApplicationContext`。



正如第 2 章中所介绍的，`@SpringBootApplication` 注解在 Spring Boot 应用程序中是一个组合了多个较低级注解的原型注解定义。`@SpringBootApplication` 中所嵌套的其中一个低级注解，就是 `@SpringBootConfiguration` 注解。它用来帮助 `@SpringBootTest` 类找到应用程序的启动类，并加载正确的 `ApplicationContext`。

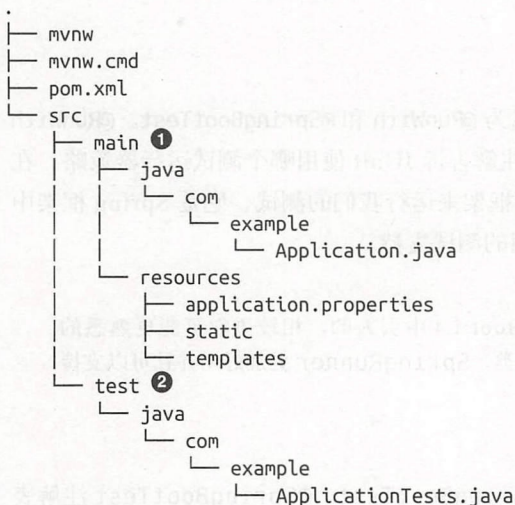
在示例 4-2 中，我们可以看到一个基本的 Spring Boot 应用程序源代码的目录结构。注意，应用程序有两个源代码根目录，两个代码包的结构相同。为了使 `ApplicationTests.java` 文件中的测试方法能够成功地扫描和查找包含在 `Application.java` 中的 Spring Boot 应用程序启动类（使用 `@SpringBootApplication` 注解），每个源代码目录的包结构应该是相同的。

`Application.java` 是一个带有 `@SpringBootApplication` 注解的类，其中包含一个简单的 `main(String[] args)` 方法。

现在我们已经介绍了，如何使用 Spring Boot 的测试依赖来创建一个自动配置的 JUnit 测试类。然后我们可以开始设计和创建集成测试，测试在 `ApplicationContext` 中加载的各个组件之间的交互效果。



示例4-2 一个新的Spring Boot工程的基础目录结构



- ❶ 源代码根目录下包含了应用程序的启动类。
- ❷ 测试源代码根目录下包含应用程序的测试类。

集成测试

正如前文提到的，需要为 Spring Boot 应用程序创建两种类型的测试：单元测试和集成测试。两种测试类型之间的主要区别在于，集成测试需要使用 Spring 上下文来测试不同组件之间的集成，而单元测试则可以测试不依赖于 Spring 库的单个组件。在本节中，我们将重点关注如何在 Spring Boot 应用程序中编写集成测试。

Spring Boot 的自动配置可能会使编写集成测试和单元测试变得困难，因为你很难将随环境变化的东西隔离开来。

十二要素程序其中一个重要的原则是，应该尽量减少开发环境和生产环境之间的差异。对于 Spring Boot 以外的大多数应用程序框架，在构建云原生应用程序时，需要严格遵循这个原则。但是对于云原生的 Spring Boot 应用程序来说，在执行关于开发环境和生产环境之间的差异规则上可能会有所偏差，因为自动配置允许我们将外部依赖替换为模拟的依赖。事实上，的确有一个地方，可以在与生产环境所使用的后端服务一样的环境中，对应用程序进行端到端的测试，我们会在第 15 章对此进行进一步讨论。



测试切片

在 Spring Boot 1.4 发布之前，需要加载了完整的 Spring 应用程序上下文之后才能执行集成测试。很多时候，这是自动配置所带来的副作用，因为并不是所有的集成测试都需要访问完整的 `ApplicationContext`。

例如，只想测试 Java 对象与 JSON 之间的序列化和反序列化，在这种情况下，几乎不需要加载 *Spring MVC* 或 *Spring Data JPA* 的自动配置。如果我们只测试 JSON 序列化，真的需要加载一个 servlet 容器吗？（答案是不。）Spring Boot 1.4 或更高版本支持测试分片的概念，即选择性激活不同层的自动配置。

测试分片还能够干净地清除某些启动器项目，例如，如果从 *Spring Data JPA* 切换到 *Spring Data MongoDB*，使用测试分片不会影响与 Spring Data 无关的集成测试。除了测试切片之外，Spring Boot 1.4 还提供了很多对编写集成测试类的支持，允许开发人员使用注解，采用声明的方式来模拟 Spring 上下文中的组件。

测试中的 Mock

我们通常在单元测试的环境下讨论 Mock。从最简单的意义上讲，Mock 对象允许我们隔离测试系统的一部分，通过一种受控的方式，用具有类似测试行为的对象来代替实际的对象。如果一个应用程序的某个服务模块需要访问某个外部后端服务，例如可能调用另一个微服务，那么我们可以插入该后端服务的一个假的实现，这样它就变成了一个确定项。唯一不确定的是被测试组件与 Mock 服务之间的交互方式。

在谈论集成测试和单元测试之间的差异时，Spring Boot 中的 Mock 有一些细微的不同。在 Spring Boot 应用程序中，集成测试中的 Mock 与单元测试一样有用。在 Spring Boot 中编写集成测试时，可以只选择 mock 测试类 `ApplicationContext` 中的指定组件。这个特性允许开发人员在测试交互组件集成的同时，还能够在应用程序的边界上 mock 对象，而单元测试和集成测试之间的区别仍然是否使用 Spring 上下文。这个区别需要时刻牢记，它有助于编写单元测试和集成测试的团队成员之间的沟通。

Spring Boot 支持 `@MockBean` 注解。`@MockBean` 会要求 Spring 在应用程序上下文中提供一个 bean 的 mock 对象，并且将上下文中原本有效的对象定义为静音。它会为 `ApplicationContext` 中的 bean 创建一个 Mockito 的 mock 对象。在示例 4-3 中，我们会看到 `AccountServiceTests` 标注了一个 `@RunWith (SpringRunner.class)` 注解，表示在它的测试执行期间不会加载 `ApplicationContext`。在这个类中，测试的组件是 `AccountService` 这个 bean，它依赖于 `UserService` bean 来访问外部的一个微服务。



示例4-3 使用@MockBean注解来mock UserService和AccountRepository组件

```
package demo.account;

import demo.user.User;
import demo.user.UserService;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.Collections;
import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.given;

@RunWith(SpringRunner.class)
public class AccountServiceTests {

    @MockBean
    ❶
    private UserService userService;

    @MockBean
    private AccountRepository accountRepository;

    private AccountService accountService;

    @Before
    public void before() {
        accountService = new AccountService(accountRepository, userService); ❷
    }

    @Test
    public void getUserAccountsReturnsSingleAccount() throws Exception {
        given(this.accountRepository.findAccountsByUsername("user")).willReturn(
            Collections
                .singletonList(new Account("user", new AccountNumber("123456789")))); ❸

        given(this.userService.getAuthenticatedUser()).willReturn(
            new User(0L, "user", "John", "Doe")); ❹

        List<Account> actual = accountService.getUserAccounts(); ❺

        assertThat(actual).size().isEqualTo(1);
        assertThat(actual.get(0).getUsername()).isEqualTo("user");
        assertThat(actual.get(0).getAccountNumber()).isEqualTo(
            new AccountNumber("123456789"));
    }
}
```



- ❶ 为 UserService 组件创建一个 Mockito 对象。
- ❷ 将 mock 出来的组件作为参数，创建一个 AccountService 的新实例。
- ❸ 使用一个账户列表，来模拟对 repository 的方法 findAccountsByUsername(String username) 的调用。
- ❹ 使用 User 的新实例模拟调用方法 getAuthenticatedUser()。
- ❺ 使用已定义的 mock 对象，来调用 AccountService 的 getUserAccounts() 方法。

在这个示例中，mock 了那些后续需要进行集成测试的协作组件的行为。在这个单元测试中，我们不需要真正远程 HTTP 调用 UserService bean 中的后端服务，就能够测试 AccountService 的功能。同样的方法也适用于 AccountRepository 组件。通常情况下，该 repository 组件被定义为 ApplicationContext 中一个自动配置的 bean，它提供了 Account 实体（映射为关系型数据库中的一张表）的数据管理功能。为了降低测试 AccountService 功能的复杂性，我们可以为 AccountRepository 组件创建一个 mock 对象，并指定它在测试时的预期行为。现在让我们来看一下 AccountService 中的内容，以更好地理解它的作用。

示例 4-4 中定义了我们正在为其编写测试的 AccountService 类。

示例4-4 依赖于相关组件的AccountService bean定义

```
package demo.account;

import demo.user.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Collections;
import java.util.List;
import java.util.Optional;

@Service
public class AccountService {

    private final AccountRepository accountRepository;

    private final UserService userService; ❶

    @Autowired
    public AccountService(AccountRepository ar, UserService us) { ❷
        this.accountRepository = ar;
        this.userService = us;
    }
}
```



```

public List<Account> getUserAccounts() {
    ③
    return Optional.ofNullable(userService.getAuthenticatedUser())
        .map(u -> accountRepository.findAccountsByUsername(u.getUsername()))
        .orElse(Collections.emptyList());
}
}

```

- ① 用于 UserService bean 的字段，会通过构造函数注入。
- ② 基于构造函数的注入，会根据每个参数将 bean 注入 ApplicationContext 中。
- ③ getAuthenticatedUser() 方法会通过远程 HTTP 调用访问用户的微服务。

值得重申的是，AccountService 使用的是构造函数注入。虽然我们可以使用字段注入，但是那样我们就会很难理解，组件为了构建一个有效的对象所期望的前提条件。请始终使用基于构造函数的注入，而不是基于字段的注入。一般来说这样做都是对的，但在测试时就不太一样了。

在 AccountService 中，Spring ApplicationContext 提供了一个 AccountRepository 的引用（如果可用的话）。然后创建了一个新的 AccountService 实例，并直接使用 AccountRepository mock 出来的对象引用来初始化该类，从而覆盖默认的行为。现在我们的测试类更像一个单元测试了——它成功地将 AccountService 组件与它的协作依赖隔离开来。

我们来看一下 UserService 组件。在它的定义中（如示例 4-5 所示），UserService 类包含了一个用来获取已认证用户的方法，它可以通过远程 HTTP 调用来访问一个后端服务的 REST API。

示例4-5 UserService bean通过远程HTTP调用访问外部的微服务

```

package demo.user;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.net.URI;

import static org.springframework.http.MediaType.APPLICATION_JSON_VALUE;
import static org.springframework.http.ResponseEntity.get;

@Service

```




```
public class UserService {

    private final String serviceHost;

    private final RestTemplate restTemplate;

    @Autowired
    public UserService(RestTemplate restTemplate,
        @Value("${user-service.host:user-service}") String sh) {
        this.serviceHost = sh;
        this.restTemplate = restTemplate;
    }

    public User getAuthenticatedUser() {
        URI url = URI.create(String.format("http://%s/uaa/v1/me", serviceHost));
        RequestEntity<Void> request = get(url).header(HttpHeaders.CONTENT_TYPE,
            APPLICATION_JSON_VALUE).build();
        return restTemplate.exchange(request, User.class).getBody(); ❶
    }
}
```

❶ 通过远程 HTTP 调用访问指定 URL 并返回一个用户的实例。

在这个相当简单的 `UserService` 组件中，我们使用 `RestTemplate` 通过 HTTP 协议向远程依赖项发出 GET 请求。因为远程资源在构建 / 测试环境中可能不可用，所以我们通过为 `AccountServiceTests` 类中定义的 `UserService` 对象，创建 `getAuthenticatedUser` 方法的 mock 和 stub，来消除对外部服务的依赖，并进一步隔离被测试的系统。

`@MockBean` 注解对用来集成测试 Spring MVC 的控制器类非常有价值，这种类通常依赖于多个协作组件，而在微服务常见的集成测试中，这些组件必须使用 HTTP 与远程服务集成。因为我们需要一个 Web 环境来测试 Spring MVC 控制器类，所以需要 `ApplicationContext`。在这种情况下，我们只希望为那些调用远程应用程序的服务创建 mock 对象。通过在 Web 应用程序的边界上 mock 对象，可以将测试组件与远程的 Web 服务依赖进行隔离。这样我们就能够在 JVM 的范围内，对应用程序模块之间的通信进行集成测试了。

对于需要 Web 环境的集成测试，`@SpringBootTest` 注解提供了额外的可配置参数，优化了在 servlet 环境下运行的测试。

使用 @SpringBootTest 中的 Servlet 容器

正如本章前面所提到的，Spring Boot 提供了多个用于测试的注解，可以帮助你测试自动配置类的指定切片。



在本章的开始，我们演示了一个使用 `@SpringBootTest` 在 Spring Boot `ApplicationContext` 上执行集成测试的例子。在大多数情况下，Spring Boot 应用程序都需要访问一个 servlet 容器，即使只是暴露一个健康指标的 HTTP 端点。一般来说，如果需要一个 servlet 容器，我们会将 Spring MVC 应用程序编译为 `.war` 文件，并将其部署到运行的应用程序服务器上。而现在对于 Spring Boot 应用程序，我们推荐使用嵌入式的 `.jar` 部署方式。

当你希望针对 Spring Boot 应用程序中，一个已经完全配置好的 `ApplicationContext` 编写集成测试时，应该使用 `@SpringBootTest` 注解。这个注解还将允许你为测试上下文配置 servlet 环境。`@SpringBootTest` 支持通过 `webEnvironment` 属性，来指定如何配置运行时使用的嵌入式 servlet 容器。表 4-1 总结了 `@SpringBootTest` 的 `webEnvironment` 属性的一些参数。

表4-1 `@SpringBootTest`的`webEnvironment`属性

选项	描述
MOCK	加载一个 <code>WebApplicationContext</code> 并提供一个 mock servlet 环境
DEFINED_PORT	加载一个 <code>EmbeddedWebApplicationContext</code> ，并在指定端口上提供一个真正的 servlet 环境
RANDOM_PORT	加载 <code>EmbeddedWebApplicationContext</code> ，并在随机端口上提供一个真正的 servlet 环境
NONE	使用 <code>SpringApplication</code> 来加载 <code>ApplicationContext</code> ，但不提供任何 servlet 环境 (mock 或其他)

最终，它决定了运行测试的延迟时间。这是对结果可信度和迭代速度之间的折衷选择。

在大多数情况下，集成测试可能不需要一个完整的 servlet 环境。每个测试类都需要重新加载 servlet 容器，才能运行其中包含的测试用例。虽然启动单个 servlet 容器所需的时间与构建 / 测试环境有关，但执行每个集成测试用例所花费的总时间可能不会太长。在持续交付和微服务的世界中，构建时间很容易成为一个限制约束。

测试分片

Spring Boot 提供了多个测试注解，可以用于测试应用程序的特定分片。

@JsonTest

`@JsonTest` 注解允许你只启用测试 JSON 序列化和反序列化的配置。我们来看一个例子。在这个例子中，我们将看到用 `@JsonTest` 注解标注的 `UserTests` 类。这个类会测试 `User` 对象是如何被序列化和反序列化的（如示例 4-6 所示）。

示例4-6 使用@JsonTest测试JSON序列化和反序列化

```
package demo.user;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.json.JsonTest;
import org.springframework.boot.test.json.JacksonTester;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@JsonTest
public class UserTests {

    private User user;

    @Autowired
    ❶ private JacksonTester<User> json;

    @Before
    public void setUp() throws Exception {
        User user = new User("user", "Jack", "Frost", "jfrost@example.com");
        user.setId(0L);
        user.setCreatedAt(12345L);
        user.setLastModified(12346L);

        this.user = user;
    }

    @Test
    public void deserializeJson() throws Exception {
        String content = "{ \"username\": \"user\", \"firstName\": \"Jack\", \""
            + \"lastName\": \"Frost\", \"email\": \"jfrost@example.com\"}";

        assertThat(this.json.parse(content)).isEqualTo(
            new User("user", "Jack", "Frost", "jfrost@example.com"));
        assertThat(this.json.parseObject(content).getUsername()).isEqualTo("user");
    }

    @Test
    public void serializeJson() throws Exception {
        ❷
        assertThat(this.json.write(user)).isEqualTo("user.json");
        assertThat(this.json.write(user)).isEqualToJson("user.json");
        assertThat(this.json.write(user)).hasJsonPathStringValue("@.username");
        ❸
    }
}
```

```

assertJsonPropertyEquals("@.username", "user");
assertJsonPropertyEquals("@.firstName", "Jack");
assertJsonPropertyEquals("@.lastName", "Frost");
assertJsonPropertyEquals("@.email", "jfrost@example.com");
}

private void assertJsonPropertyEquals(String key, String value)
    throws java.io.IOException {
    assertThat(this.json.write(user)).extractingJsonPathStringValue(key)
        .isEqualTo(value);
}
}

```

- ❶ 由 Jackson 提供的基于 AssertJ 的 JSON 测试程序。
- ❷ 将 User 对象写入 JSON 并与 user.json 文件进行比较。
- ❸ 断言实际的 JSON 结果是否与预期的属性值相匹配。

对于断言 JSON 序列化的实际结果，是否与测试方法的预期结果相匹配，有很多的选项可以使用。如果要将类路径上的 JSON 文件映射为一个测试资源，需要在测试方法内部做很多清理工作，尤其是当预期的 JSON 结果有许多属性时。一种用来比较的方法，是在测试代码中使用笨重的、硬编码的 JSON 字符串，例如 `deserializeJson` 方法所示。

测试用例 `serializeJson` 使用了另一种方法，即试图将 `setUp` 方法中初始化的 User 对象，序列化成 JSON 字符串。在这个方法中，引用了一个 `user.json` 资源，作为 `isEqualTo` 方法的一个参数。为了测试 Jackson JSON 序列化生成的预期结果，还指定了一个 JSON 文件作为类路径资源。同测试的包目录结构一样，`.json` 文件位于 `src/main/resources` 目录中（如示例 4-7 所示）。

示例4-7 包含UserTests类的测试源代码根目录结构

```

├─ ./src/test
│
├─ java
│   └─ demo
│       └─ user
│           ├── UserControllerTest.java
│           ├── UserRepositoryTest.java
│           └─ UserTests.java
│
└─ resources
    ├── data-h2.sql
    └─ demo
        └─ user
            └─ user.json *

```

我们来看一下 `user.json` 的内容（如示例 4-8 所示）。

示例4-8 测试资源中user.json文件的内容

```
{
  "username": "user",
  "firstName": "Jack",
  "lastName": "Frost",
  "email": "jfro@example.com",
  "createdAt": 12345,
  "lastModified": 12346,
  "id": 0
}
```

@WebMvcTest

@WebMvcTest 注解支持在 Spring Boot 应用程序中，测试各个 Spring MVC 的控制器。这个注解自动配置了用于测试与控制器方法的交互所需的 Spring MVC 基础环境（如示例 4-9 所示）。

示例4-9 使用@WebMvcTest测试Spring MVC控制器

```
package demo.account;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Collections;

import static org.mockito.BDDMockito.given;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*
    .get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*
    .content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*
    .status;

@RunWith(SpringRunner.class)
@WebMvcTest(AccountController.class)
public class AccountControllerTest {

    @Autowired
    private MockMvc mvc; ❶
```

```

@Bean
private AccountService accountService; ❷

@Test
public void getUserAccountsShouldReturnAccounts() throws Exception {
    String content = "[{\"username\": \"user\", \"accountNumber\": \"123456789\"}]";

    ❸
    given(this.accountService.getUserAccounts()).willReturn(
        Collections.singletonList(new Account("user", "123456789")));

    ❹
    this.mvc.perform(get("/v1/accounts").accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk()).andExpect(content().json(content));
}
}

```

- ❶ 模拟 MVC 客户端对 Spring MVC 控制器执行 HTTP 请求。
- ❷ 模拟 AccountController 调用的 AccountService 组件。
- ❸ 定义从 accountService bean 中获取用户账户的预期行为。
- ❹ 最后，使用 MockMvc 客户端来断言 AccountController 期望的 HTTP 结果。

AccountControllerTest 演示了如何使用 MockMvc 客户端，向被测试的 Spring MVC 控制器执行模拟的请求。MockMvc 来自 Spring MVC 测试框架。如示例中所配置的测试，展示了 Spring MVC 的全部机制（除了一个真实的 servlet 容器以外），并且将请求从客户端路由到控制器本身。控制器会生成一个响应，然后将其发送回客户端。这几乎就像你通过电话拨打实际的服务一样，只不过不是 HTTP 服务而已。实际上在 ServerSocket 上不会发送和处理任何请求。

@DataJpaTest

Spring Data 从 1.4 开始，提供了一个新的 @DataJpaTest 测试注解。这个注解对使用了 Spring Data JPA 项目的 Spring Boot 应用程序非常有用。它为 Spring Data JPA 在测试时提供了嵌入式的内存数据库支持。在第 9 章讨论数据访问时，介绍如何在集成测试中配置不同的 Spring profile。现在，我们先来了解一下，如何在 MySQL 和 H2（一种嵌入式的内存关系数据库）之间进行运行时切换。

在示例 4-10 中，定义了一个名为 AccountRepositoryTest 的测试类，并使用 @DataJpaTest 注解进行标注。只有在这段测试分片下，用来执行 Spring Data JPA repository 上测试所需的自动配置类才会被激活。在测试方法中，我们使用了 TestEntityManager，这是 Spring Boot 提供的一个工具类，支持 JPA EntityManager 中一部分有用的功能子集，以

及一些额外的、解决测试中常见问题的工具方法。TestEntityManager 是 JPA repository 测试中一个很有用的组件，它允许不使用实际的 repository，就可以使用底层的数据存储来存储对象。这里，我们用它来存储一个 Account 实体的新实例，表示期望从 accountRepository 返回的实际结果。

示例4-10 使用@DataJpaTest测试Spring Data JPA repository

```
package demo.account;

import demo.customer.CustomerRepository;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@DataJpaTest
public class AccountRepositoryTest {

    private static final AccountNumber ACCOUNT_NUMBER = new AccountNumber(
        "098765432");

    @Autowired
    private AccountRepository accountRepository; ❶

    @Autowired
    private TestEntityManager entityManager; ❷

    @Test
    public void findUserAccountsShouldReturnAccounts() throws Exception {
        this.entityManager.persist(new Account("jack", ACCOUNT_NUMBER)); ❸
        List<Account> account = this.accountRepository.findAccountsByUsername("jack"); ❹
        assertThat(account).size().isEqualTo(1);
        Account actual = account.get(0);
        assertThat(actual.getAccountNumber()).isEqualTo(ACCOUNT_NUMBER);
        assertThat(actual.getUsername()).isEqualTo("jack");
    }

    @Test
    public void findAccountShouldReturnAccount() throws Exception {
        this.entityManager.persist(new Account("jill", ACCOUNT_NUMBER));
        Account account = this.accountRepository
            .findAccountByAccountNumber(ACCOUNT_NUMBER);
        assertThat(account).isNotNull();
        assertThat(account.getAccountNumber()).isEqualTo(ACCOUNT_NUMBER);
    }
}
```

```

@Test
public void findAccountShouldReturnNull() throws Exception {
    this.entityManager.persist(new Account("jack", ACCOUNT_NUMBER));
    Account account = this.accountRepository
        .findAccountByAccountNumber(new AccountNumber("0000000000"));
    assertThat(account).isNull();
}
}

```

- ❶ 从 ApplicationContext 中注入 AccountRepository。
- ❷ 注入 TestEntityManager，以便能够在不使用 repository 的情况下存储对象。
- ❸ 将新的 Account 实体对象，存储到为上下文配置的内存数据库中。
- ❹ 找到已存储的 Account 对象。

@RestClientTest

@RestClientTest 注解为测试 Spring 的 RestTemplate，以及 RestTemplate 与 REST 服务的交互提供了支持。

在示例 4-11 中，我们可以看到一个用 @RestClientTest 标注的单元测试。指定目标为 UserService 类，并且希望 RestTemplate 被注册到自动配置的测试分片中。在测试方法 getAuthenticatedUserShouldReturnUser() 中，使用 MockRestServiceServer 字段 (server) 来模拟 UserService 中 RestTemplate 发起的 HTTP 请求。注意预期的 JSON 响应，是如何从类路径资源 user.json 加载的。

示例4-11 使用@RestClientTest模拟RestTemplate响应

```

package demo.user;

import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;

//@formatter:off
import org.springframework.boot.test.autoconfigure.web.client.AutoConfigureWebClient;
import org.springframework.boot.test.autoconfigure.web.client.RestClientTest;
//@formatter:on

import org.springframework.core.io.ClassPathResource;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.client.MockRestServiceServer;

```



```
import static org.assertj.core.api.Assertions.assertThat;

//@formatter:off
import static org.springframework.test.web.client.match
    .MockRestRequestMatchers.requestTo;
import static org.springframework.test.web.client.response
    .MockRestResponseCreators.withSuccess;
//@formatter:on
```

❶

```
@RunWith(SpringRunner.class)
@RestClientTest({ UserService.class })
@AutoConfigureWebClient(registerRestTemplate = true)
public class UserServiceTests {
```

```
    @Value("${user-service.host:user-service}")
    private String serviceHost;
```

```
    @Autowired
    private UserService userService;
```

```
    @Autowired
    private MockRestServiceServer server;
```

```
    @Test
    public void getAuthenticatedUserShouldReturnUser() {
        this.server.expect(
            requestTo(String.format("http://%s/uaa/v1/me", serviceHost))).andRespond(
                withSuccess(new ClassPathResource("user.json", getClass()),
                    MediaType.APPLICATION_JSON)); ❷
```

```
        User user = userService.getAuthenticatedUser();
```

```
        assertThat(user.getUsername()).isEqualTo("user");
        assertThat(user.getFirstName()).isEqualTo("John");
        assertThat(user.getLastName()).isEqualTo("Doe");
        assertThat(user.getCreatedAt()).isEqualTo(12345);
        assertThat(user.getLastModified()).isEqualTo(12346);
        assertThat(user.getId()).isEqualTo(0L);
    }
}
```

❶ 为自动配置的测试分片的上下文注册一个 RestTemplate。

❷ 模拟 RestTemplate 向指定 URL 发送请求的行为。

在使用 MockRestServiceServer 模拟来自远程服务的 HTTP 响应之后，任何使用 RestTemplate 来匹配预期值的请求，都将返回 user.json 的内容。这样做的好处是，我们可以通过模拟外部服务的行为，来对部分系统进行测试。这是一个非常实

用的功能，特别有助于我们测试使用 HTTP 互相进行调用的微服务系统。接下来的问题是，我们如何创建集成测试来模拟远程 HTTP 服务的实际行为，而不是使用 MockRestServiceServer？稍后会介绍这个主题，它允许我们下载其他服务发布的存根 (Stub)，以便模拟集成测试中的服务行为。

端到端测试

端到端测试是一种确保分布式应用程序，在更改组件后不影响用户使用的重要手段。此外，微服务架构由许多不同的服务组成，每个应用程序可能会连接多个服务，形成一个测试复杂的、有组织的混乱集合。本节我们将介绍一些测试分布式云原生应用程序的技术。

端到端测试侧重于验证应用程序的业务功能。与集成测试相反，端到端测试侧重于从用户角度测试功能。例如，我们假设某个应用程序的用户想要注册一个新账户。用户必须执行一系列操作才能成功注册新账户。在大多数情况下，应用程序的用户界面会将这些操作连接起来，产生与后端 API 交互的事件来完成整个工作流程。如果我们对用户进行注册新账户操作时，产生的每个事件进行清点，我们可以将这一系列事件转换为端到端测试。

根据对系统测试的部分不同，端到端测试有多种类型。对于构建微服务的后端开发人员来说，注册新账户的端到端测试可能是由不同微服务之间的 API 交互所组成。对于一个涉及不同微服务的业务功能，如何编排一个端到端的工作流，以及如何管理状态就成为重要的考虑要素。

测试分布式系统

当我们谈论软件的一致性时，通常是指状态。在分布式系统中开发应用程序时，需要准确了解在架构之中如何共享和复制状态。一种我们最喜欢的状态解释会带我们一直回到计算的诞生时代。图灵首先在他的开创性论文“论可计算数及其在判定问题上的应用”中提到了计算状态。在该论文中，图灵用一个思维实验来描述状态，该实验可以从理论上解释一个机器如何确定一个数字是否是可计算的。图灵的思想实验引入了一个图灵机^{注1}的概念，即一个想象中的能够使用状态表来计算数字的机械设备。

图灵描述的机器包含一个由方块组成的无限磁带，其中方块中包含仅在两个方向上伸展的符号。图灵将他的机器简单地类比成打字机。机器可以在磁带上向左或向右滚动，一次只能扫描一个符号。机器还有一个状态表，用来将符号映射为指示机器下一步该做什

注1 “论可计算数及其在判定问题上的应用” (<http://bit.ly/2s6w8G0>)，伦敦数学学会，1937 年。

么的条件。

我们来看一个例子。我们假设图灵的机器处于初始状态 A。机器从磁带扫描的第一个符号为 0。然后，机器从状态表中查找其当前状态的指令，该状态仍为 A。指令描述了一个条件：如果扫描到符号 0，写入 1。在移动到下一个位置之前，最后的指令是从 A 到 B 的状态改变。根据下一个位置上扫描到的符号，状态 B 会包含不同的指令集。现在机器可能再次从新的位置扫描到 0，并且因为它处于状态 B，它仍将写入 1，但是这次机器被要求在磁带上左移，而不是右移。

图灵通过切换根据不同输入而改变的指令，来输出数据，从而实现了图灵机的编程。

当我们讨论现代应用程序中的状态时，我们通常指的是数据库记录上的状态字段，体现为表中的某一列。举个例子，某个用户在一个网站上注册了一个新账户。用户的状态会根据用户的输入从一个状态转换到另一个状态。如果用户刚刚通过提交网页表单注册了账户，则系统会处理提交的表单字段，并将记录持久保存到用来存储用户注册细节的数据库中。这个新用户记录的其中一个字段是 `status`，它在任一时刻，都会包含一个描述用户当前状态的值。有关的示例记录，参见表 4-2。

表4-2 包含状态字段的用户记录

first_name	last_name	email	status
Bob	Dylan	<i>bdylan@example.com</i>	PENDING
Taylor	Swift	<i>tswift@info.com</i>	CONFIRMED
Tracy	Chapman	<i>tchapman@test.com</i>	ARCHIVED
Bruno	Mars	<i>bmars@example.com</i>	INACTIVE

在表 4-2 中，可以看到，数据库中包含了许多条用户记录。表中每个用户的 `status` 列都有不同的值。其中名为 Bob Dylan 的用户，其当前状态是 `PENDING`。现在，根据 `status` 列的值，从用户角度看，应用程序的行为将发生改变。对于 `PENDING` 状态的用户，在用户可以登录应用程序之前，用户需要确认列出的电子邮件地址。用户确认电子邮件后，用户的状态将变为 `CONFIRMED`，此时允许用户登录并访问应用程序的其他功能。如果用户的状态设置为 `ARCHIVED` 或者 `INACTIVE`，则用户可能无法登录，具体由应用程序的需求决定。这个流程通常会用在实现用户认证的应用程序中。

分布式系统开发（更具体地说，是微服务）所带来的问题是，像上述那样的工作流程不会被隔离在单个应用程序的边界之内。当我们把一个应用程序架构分解成许多单独的服务时，网络分区会将原本在单个事务上下文中管理数据的工作流程分开。那么这对开发者意味着什么呢？

分布式应用程序的主要问题是，必须能够使用通过网络交换的信息来进行状态通信，从

而以全局视角来观察不同机器的状态。对于连接到单个数据库（例如像 MySQL 这样的关系数据库）的某个单体应用程序，状态可以在数据库的不同实例之间始终保持复制状态，从而使其具有高可用性。通过在多台机器上存储一条包含状态的记录，我们能够使用水平扩展来处理不断增加的流量。

如下问题最好地总结了分布式状态的缺点：我们如何确保，每当从一个机器池中读取一个已复制的记录时，它与池中其他机器上存储的记录的状态是一致的？

分布式状态和保证一致性是计算机科学中最难的问题之一。在单个机器和进程内，维护数据的一致性会非常简单——因为每当我们从内存中读取数据时，都只有一个地址引用指向数据的存储位置。如果一个线程在该地址上锁定引用，试图改变记录的状态，则其他线程在继续修改该引用的记录状态之前，会先观察锁的情况。当一条记录的状态只有一个单一的信息来源时，它就可以被全局访问，并且记录状态不会出现不一致的情况。当状态跨越单个机器进行复制时，这种情况会发生变化，所有副本必须被一起更新并一起读取。如果其中一个副本与其他的副本不同，则记录的状态出现了不一致，这可能会导致应用程序会根据不同分区上存储的不同状态，产生不同的行为。

在测试状态在多个微服务上分布的云原生应用程序时，你必须知道如何设计测试数据一致的端到端测试。对于在多个微服务之间共享状态的情况，我们希望最好是达到最终一致性。第 12 章在讨论数据集成解决方案时，会讨论如何设计最终一致性。现在，关键的问题是如何设计测试条件，以确保状态总是最终一致的，即可能在同一时间的状态不一致，但不需要手动干预总可以达到最终一致的状态。不要在需要强一致性的微服务之间共享状态。如果想知道何时避免使用分布式事务，这就是一个简单的经验法则。

微服务体系结构的另一个问题是，我们测试不同应用程序之间集成的方式。对于这种测试，我们通常可能需要在集成环境中执行端到端测试，而在该集成环境中，所有应用程序和依赖关系都完全参考生产环境的部署模型，在运行时才会加载和执行。这种需要引导一个完整端到端环境的时间成本，对于一个单独的微服务来说可能是极其昂贵的。

对于开发人员来说，虽然服务依赖众多，但是他们希望能够尽可能快地执行集成测试，从而创建更快的反馈循环，并限制在共享环境中的资源竞争。一种常用的测试方法有助于解决这些难题，并且正在迅速成为测试微服务的标准，这就是消费者驱动的契约测试。

消费者驱动的契约测试

消费者驱动的契约测试（CDC-T）是伊恩·罗宾逊（Ian Robinson）在 2006 年首次提出的，它是利用已发布的契约来断言和维护消费者和生产者之间的期望，同时保证服务之间的松耦合性。在最初发表在 Martin Fowler 网站上的文章中，Robinson 通过使用消费者驱

动的契约来描述面向服务架构（SOA）中的服务演进。^{注1}在接下来的几年中，人们逐渐接受了 Robinson 基于消费者驱动的契约的实践和模式，并使用它们来测试微服务之间的期望关系。

CDC-T 的核心前提是允许微服务架构中的生产者和消费者以（消费者驱动）契约的形式发布（并建立）存根。契约描述了服务的调用接口。运行中的微服务之间也不存在共享的库。但是，在微服务之间存在周期性依赖的情况下，服务可以在集成测试期间相互共享库。生产者需要首先通过定义契约以及使用契约的集成测试来发布存根（stub）。消费者可以从某个共享位置（例如包仓库）下载生产者不同版本的存根（stub），从而来模拟生产者。生产者之间可以共享通过契约定义生成的存根，但不能共享类型或客户端库。CDC-T 努力隐藏生产者的 API 的实现细节。

Spring Cloud Contract

Spring Cloud Contract 是 Spring 家族中的一个开源项目，它通过一种消费驱动契约的变体来提供框架组件。消费者驱动契约通常用于集成测试分布式的应用程序组件，例如 REST API 和微服务之间的消息交换。Spring Cloud Contract 支持使用存根发布和模拟远程服务的能力。

在本书示例中，我们将创建一个消费者驱动的契约，来测试两个微服务之间的集成。我们将测试两个服务：账户（Account）微服务和用户（User）微服务。账户微服务用来维护属于某个用户的账户集合。对于拥有这些账户的用户，用户微服务会用来存储这些用户的记录，同时负责用户的身份认证。

这两个微服务整合起来很简单。账户微服务只需要获得当前登录到网站的用户账户。为此，它会通知用户微服务，从 HTTP 请求的会话中获取已经通过身份认证的用户的名称。帐户微服务然后会根据从用户微服务返回的用户名，获取已通过身份认证的用户的账户信息。账户微服务和用户微服务之间的关系如图 4-1 所示。

注 1 Ian Robinson, “消费者驱动的契约：一种服务演化的模式” (<http://bit.ly/2s69I7x>), 2006。

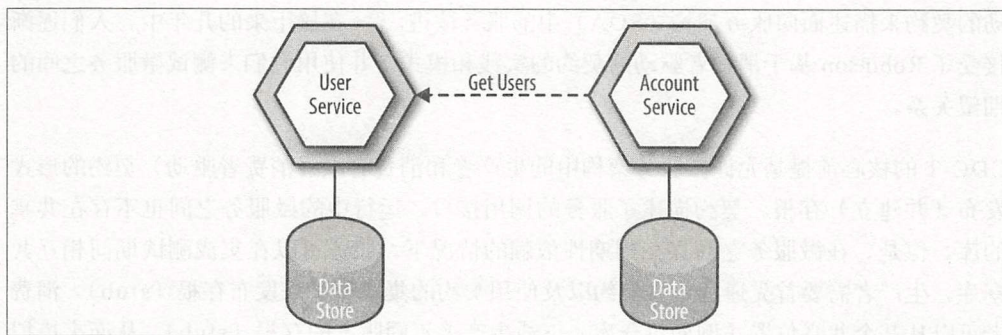
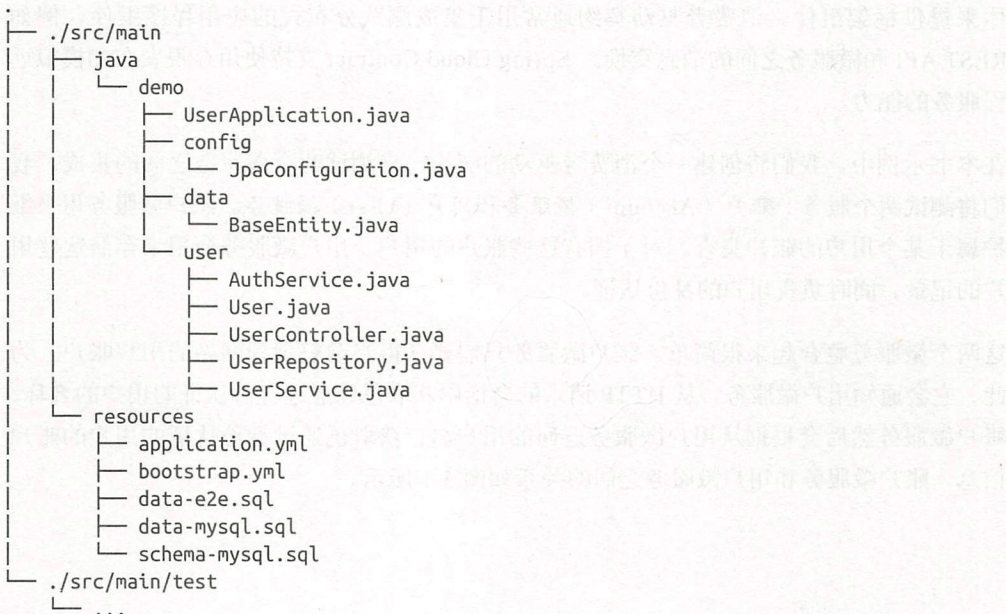


图4-1 账户服务是用户服务的消费者

现在，让我们先来看一下两个 Spring Boot 项目的源代码，首先从用户微服务（示例 4-12）开始。

示例4-12 用户微服务的Spring Boot应用程序类



示例 4-12 显示了用户微服务的 Spring Boot 应用程序源代码。这里的目录结构，主要集中在 `src/main/demo` 目录中的应用程序类文件中。该包中包含了用户微服务的应用程序源代码。由于我们已经知道一个基本的 Spring Boot 应用程序是如何工作的，所以这里我们仅探索这个服务的核心功能，即如何使用一个消费者驱动的契约进行测试。

用户微服务的 `user` 包中含有我们测试需要的功能。这里我们找到了两个含有业务逻辑

辑的服务组件：AuthService 和 UserService。UserService 中包含一个方法，用于获取会话中已通过认证的用户。AuthService 用来获取已认证用户的会话信息。然后，UserService 会使用 AuthService 返回的 ID 作为查询条件，从 UserRepository 中查找出用户的详细信息。

在示例 4-13 中，显示了一个名为 getUserByPrincipal 的方法，它返回一个 User 类的实例。User 类是一个域实体，其中多个字段用来描述已通过身份认证的 Principal 对象。该方法使用 Principal 类的 name 属性从 UserRepository 中查找 User 记录。在本示例中，我们使用 Spring Data JPA repository 来操作数据库（第 9 章会更详细地探讨 Spring Data JPA）。

示例4-13 UserService类会获取一个用户主体的详细信息

```
package demo.user;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.security.Principal;
import java.util.Optional;

@Service
public class UserService {

    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserByPrincipal(Principal principal) {
        ❶
        return Optional.ofNullable(principal)
            .map(p -> userRepository.findUserByUsername(p.getName()))
            .orElse(null);
    }
}
```

- ❶ 使用 Principal 参数的 name 字段来查找用户记录。



这个例子并不完全适用于实际场景，它展示了在微服务中实现用户安全的常见工作流程，但是没有实现身份认证的提供者。该示例中的 AuthService 只包含一个 dummy 方法，用于获取一个 java.security.Principal 的空实现。在实际的应用程序中，可以将其中的大部分工作交给 Spring Security。

我们来探讨一下 UserController 类，该类定义了一个 REST API，Account Service 将使用该 API，通过 HTTP 远程获取通过身份认证的用户。

在示例 4-14 中，给出了 UserController 类的定义。该类定义了一个简单的、映射到 /uaa/v1 端点的 Spring MVC 控制器。这里有一个叫作 me 的控制器方法。此方法试图从 HTTP 请求上下文的会话中，获取一个 Principal 对象。首先，它尝试从 AuthService 获取通过身份认证的 Principal 对象。如果 Principal 可用且不为 null，则控制器方法将调用 UserService 类，从数据库中获取用户记录。最后，如果不能获取到 User 或者 Principal 对象，则该方法会返回一个 HTTP 状态为 UNAUTHORIZED 的 ResponseEntity。

示例4-14 UserController类定义了应用程序的REST控制器

```
package demo.user;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.security.Principal;
import java.util.Optional;

@RestController
@RequestMapping(path = "/uaa/v1")
public class UserController {

    private UserService userService;

    private AuthService authService;

    @Autowired
    public UserController(UserService userService, AuthService authService) {
        this.userService = userService;
        this.authService = authService;
    }

    ❶

    @RequestMapping(path = "/me")
    public ResponseEntity<User> me(Principal principal) throws Exception {
        return Optional.ofNullable(authService.getAuthenticatedUser(principal)) ❷
            .map(p -> ResponseEntity.ok().body(userService.getUserByPrincipal(p))) ❸
            .orElse(new ResponseEntity<User>(HttpStatus.UNAUTHORIZED)); ❹
    }
}
```

❶ 描述 GET 请求 /uaa/v1 me 的 @RequestMapping。

❷ 从 AuthService 获取通过身份认证的用户 Principal 对象。

- ③ 如果 `Principal` 不为 `null`，则通过 `UserService` 查找用户记录。
- ④ 如果 `Principal` 为 `null`，则返回 HTTP 未授权错误。

接下来，我们将为映射到 `/uaa/v1/me` 端点的 `UserController` 方法，创建一个消费者驱动的契约测试。我们将使用 *Spring Cloud Contract* 来发布一个用户微服务的 REST API 规范。这个规范是通过一个 Maven 构件发布的，它可以被其他微服务获取到，用来作为一个模拟用户微服务测试行为的 Web 服务器。这样，用户微服务只需要发布一个规范，就可以通过所有为 `UserController` 编写的单元测试。通过采取这种方法，用户微服务的所有消费者能够在测试执行期间，对模拟出来的 Web 服务器进行集成测试，就好像它是真实的服务一样！

我们来看一下，为用户微服务定义了消费者驱动的契约的测试源代码。示例 4-15 给出了测试的目录结构。

示例4-15 用户微服务的测试源代码根目录

```
├── ./src/main/test/java
│   ├── demo
│   │   ├── UserServiceBase.java
│   │   └── user
│   │       ├── UserControllerTest.java
│   │       ├── UserRepositoryTest.java
│   │       └── UserTests.java
└── resources
    ├── contracts
    │   └── shouldReturnUser.groovy *
    ├── data-h2.sql
    └── demo
        └── user
            └── user.json
```

本例中用于单元测试和集成测试的测试策略，与本章前面例子中使用的策略相同。这里重点介绍如何为用户微服务创建一个 *Spring Cloud Contract* 的存根定义。我们所创建的存根定义，会定义一个消费者驱动测试，用来获取通过身份认证的用户。

Spring Cloud Contract 中的存根定义，用来描述消费者驱动测试的模拟（远程）服务行为。这些行为被称为期望，因为它们是服务生产者对于被暴露给消费者的 REST API 方法，在测试下的行为期望。

每个存根定义文件会对应一个 `Controller` 类的某个方法。由于 `UserController` 中只有一个方法，因此只会为该项目创建一个存根定义文件，名为 `shouldReturnUser.groovy`。*Spring Cloud Contract* 的存根定义使用 *Spring Cloud Contract Groovy* 语言 DSL 编写。默

认情况下，存根定义位于测试根目录的资源目录 `src/main/test/java/resources` 中。下面我们来看一下这个定义（如示例 4-16 所示）。

示例4-16 `shouldReturnUser.groovy`中的存根定义

```
package contracts

org.springframework.cloud.contract.spec.Contract.make {
    request {
        method 'GET'
        url '/uaa/v1/me'
        headers {
            header('Content-Type': consumer(regex('application/*json*')))
        }
    }
    response {
        status 200
        body([
            username      : value(producer(regex('[A-Za-z0-9]+'))),
            firstName     : value(producer(regex('[A-Za-z]+'))),
            lastName      : value(producer(regex('[A-Za-z]+'))),

            // @formatter:off
            email          : value(producer(
                regex('[A-Za-z0-9]+\@[A-Za-z0-9]+\.[A-Za-z]+')),
            // @formatter:on
            createdAt      : value(producer(regex('[0-9]+'))),
            lastModified   : value(producer(regex('[0-9]+'))),
            id             : value(producer(regex('[0-9]+')))
        ])
        headers {
            header('Content-Type': value(
                producer('application/json;charset=UTF-8'),
                consumer('application/json;charset=UTF-8')
            ))
        }
    }
}
```



如果你不熟悉 Groovy DSL 语法，不要担心！Spring Cloud Contract 中存根定义的 DSL 很简单，很容易就可以上手和学习。虽然这里不会详尽地介绍 Groovy DSL，但是你可以从 Spring Cloud Contract 项目的在线参考手册文档中找到更多的详细信息。

将 `user-microservice` 构件安装在本地 Maven 仓库或 Maven 构件仓库中。`user-microservice` 被配置为在每次 `mvn install` 时，发布两个构件（注意不是一个）：构件本身和包含契约定义的构件。这第二个构件就是我们的消费者 API（账户微服务）将依赖的东西。示例 4-17 显示了 Maven 的构建配置。


```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-contract-maven-plugin</artifactId>
      <version>${spring-cloud-contract.version}</version>
      <extensions>true</extensions>
      <configuration>
        <baseClassForTests>demo.UserServiceBase</baseClassForTests>
        <basePackageForTests>demo</basePackageForTests>
      </configuration>
    </plugin>
  </plugins>
</build>
```

检查你的本地 Maven 仓库，确认构件和构件存根已经被部署（如示例 4-18 所示）。

示例4-18 应该安装到本地Maven仓库的构件文件示例

```
.
├─ _remote.repositories
├─ maven-metadata-local.xml
├─ user-microservice-1.0.0-SNAPSHOT-stubs.jar
├─ user-microservice-1.0.0-SNAPSHOT.jar
└─ user-microservice-1.0.0-SNAPSHOT.pom
```

0 directories, 5 files

在部署了存根之后，可以在消费者测试中，使用 stub runner 来代替符合契约并且可以与消费者集成的 REST API。这个 API 是真实的，它在一个实际的端口上运行，并根据规范的定义产生实际的值。运行这个测试会很简单和快速，因为不需要处理服务与中间件和数据库之间的交互（如示例 4-19 所示）。

示例4-19 账户服务的单元测试，可以测试用户服务的消费者驱动契约

```
package demo;

import demo.user.User;
import demo.user.UserService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

//@formatter:off
import org.springframework.cloud.contract.stubrunner
    .spring.AutoConfigureStubRunner;
//@formatter:on
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;
import static org.springframework.boot.test.context.SpringBootTest.*;

//@formatter:off
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@AutoConfigureStubRunner(
    ids = { "cnj:user-microservice::stubs:8081" },
    workOffline = true) ❶

//@formatter:on
public class ConsumerDrivenTests {

    @Autowired
    private UserService service; ❷

    @Test
    public void shouldReturnAuthenticatedUser() {
        User actual = service.getAuthenticatedUser();

        assertThat(actual).isNotNull();
        assertThat(actual.getUsername()).matches("[A-Za-z0-9]+");
        assertThat(actual.getFirstName()).matches("[A-Za-z]+");
        assertThat(actual.getLastName()).matches("[A-Za-z]+");
        assertThat(actual.getEmail()).matches(
            "[A-Za-z0-9]+\\@[A-Za-z0-9]+\\. [A-Za-z]+");
    }
}
```

- ❶ @AutoConfigureStubRunner 注解指定了去哪里找到存根工件（按照 Maven 的指示）以及 stub runner 应该在哪个端口上运行模拟服务。
- ❷ 单元测试注入了实际的 UserService。这里没有对服务进行任何 mock，只是在调用存根服务后会收到相应的响应信息！

总结

正如我们在本章中所讨论的，微服务的测试方式可能与其他 Web 应用程序的单元测试和集成测试有很大不同。Spring 为测试 Spring Boot 应用程序提供了一组功能强大的框架组件和项目。

Spring 已经为单个应用程序中的集成测试提供了强大的支持。Spring Boot 对其进行了扩展，使得在多层功能之间区分配置变得更加容易。Spring Cloud 通过消费者驱动契约测试（CDC-T）又进一步进行了扩展。在本章中，我们讨论了如何使用 Spring Cloud Contract 来模拟 REST 端点。值得一提的是，Spring Cloud Contract 也支持基于消息的端点。

测试是持续交付的一个重要方面。在持续交付中，软件构件会通过一个管道向后移动，最终产生可用于生产环境的构件。随着软件在持续交付管道中不断前进，测试会变得越来越复杂和缓慢。理想情况下，单元测试和集成测试应该在非常短的时间内完成 80% ~ 90% 的用例，从而确保此条件下的代码（可能）可以被推向生产环境。

在本章中，我们并没有把重点放在单元测试上，也没有关注集成测试之后更复杂但不常见的测试。在第 2 章中，我们已经了解了如何使用 Cloud Foundry Java 客户端将应用自动部署到 Cloud Foundry。在云原生系统中，一切都是自动的，包括部署。这是有效完成冒烟测试和获得一致性的关键因素。

迁移遗留的应用程序

假设现在你已经有了新的分布式运行时环境、无限的潜力和大量现成的应用程序。接下来该怎么办？

契约

Cloud Foundry 旨在通过减少部署和管理应用程序的相关运维问题，或者至少让它们保持一致，从而提高速度。Cloud Foundry 是运行在线 Web 服务和应用程序、服务集成以及后台处理的理想场所。

Cloud Foundry (<http://cloudfoundry.org>) 通过假设应用程序的运行形状，来优化 Web 应用程序和服务的持续交付过程。Cloud Foundry 的输入是应用程序：Java（.jar 或者 .war）二进制文件、Ruby on Rails 应用程序、Node.js 应用程序等。Cloud Foundry 为在它之上运行的应用程序，提供了众所周知的运维优势（日志聚合、路由、自我修复、动态扩展和收缩、安全等方面）。平台和应用程序之间有一个隐含的契约，这个契约要求平台保证它对其中所运行应用程序的承诺。

有些应用程序可能永远无法满足该契约。其他应用程序也许能够，但是需要进行一些细微的调整。在本章中，我们将研究一些可能采用的微重构手段，以便让那些遗留的应用程序在 Cloud Foundry 上运行。

在这种情况下，我们的目标不是构建云原生的应用程序，而是将现有工作迁移到云端，以减少企业的运维成本，提高统一性。一旦我们在 Cloud Foundry 上部署了一个应用程序，它应该至少同以前一样正常运行，因此现在只需要进行雪花式部署，即每次部署时都仅有少量改动。这体现了少即是多的思想。

我们将这种构建云原生应用程序所需的工作迁移，称为应用程序迁移。我们在 Spring

(<http://spring.io/blog>) 和 Pivotal (<http://pivotal.io/blog>) 博客上讨论的大部分内容都是关于如何构建云原生应用程序的——那些在云上生存和呼吸的应用程序（它们根据需求和容量进行扩容或者收缩）和那些充分利用平台能力的应用程序。不过，这个过程并没有理想中那么简单，需要进行更长时间、更大范围的讨论，因此这不是本章的重点。但是，这绝对是本书所有其他章节的重点。

一般来说，应用程序的行为是指它的环境和代码的总和。在本章中，我们将介绍一些如何从某些传统环境中迁移遗留 Java 应用程序的方法。还将介绍在云计算到来之前，开发传统应用程序所使用的模式，以及在云上运行应用程序最合适的模式。我们将展示一些专业的解决方案及相关代码。

迁移应用程序环境

所有应用程序都有一些共同的特性，这些特性（例如 RAM 和 DNS 路由）可以直接通过 Cloud Foundry 的 cf CLI 工具、各种仪表板或应用程序的 `manifest.yml` 文件进行配置。如果你的应用程序只需要更多的 RAM 或自定义的 DNS 路径，那么你在 Cloud Foundry 上已经可以实现它所有的基本操作了。

开箱即用的构建包 (Buildpacks)

不过事情有时候并不会那么简单。你的应用程序可能运行在多个不一样的环境中，而 Cloud Foundry 会对应用程序运行的环境做出非常明确的假设。这些假设在平台本身和构建包（从 Heroku 中借鉴而来）中都有一定的编码。Cloud Foundry 和 Heroku 并不关心它们正在运行什么样的应用程序。它们关心的是 Linux 容器，最终会是操作系统进程。构建包会告诉 Cloud Foundry 如何处理 Java .jar、Rails 应用程序、Java .war 和 Node.js 应用程序等，以及如何将其转换为平台可以像处理进程一样处理的容器。构建包实际上是一组回调方法（即可以响应已知调用的 shell 脚本），运行时环境会通过构建包来最终创建一个运行的 Linux 容器。这个过程被称为预发布 (staging)。

Cloud Foundry 提供了许多开箱即用的系统构建包 (<http://docs.pivotal.io/pivotalcf/buildpacks>)。这些构建包可以被定制甚至被完全替换掉。如果你想要运行一个没有提供任何现成构建包的应用程序，如 Cloud Foundry 社区 (<http://bit.ly/2s6o0Fe>)、Heroku (<https://www.heroku.com>) 或 Pivotal (<https://pivotal.io>) 都没有提供，那么至少你可以很容易地开发和部署自己的构建包 (<http://docs.pivotal.io/pivotalcf/buildpacks/custom.html>)。现在已经有了能够用于各种环境和应用程序的构建包，包括一个名为 Sourcey (<https://github.com/oetiker/sourcey-buildpack>) 的编译器，它只是用来帮你编译本地代码。（不得不承认，你一定知道某个业务线上遗留的应用程序，还需要一个 C 编译器和一些

复杂的编译步骤，不是吗？)

自定义的构建包

这些构建包的目的在于提供合理的默认设置，同时保持灵活性。例如，默认的 Java / JVM 构建包支持 .war 文件 (<https://github.com/cloudfoundry/java-buildpack>) (它将运行在最新版本的 Apache Tomcat 中)、Spring Boot 风格的 .jar 可执行文件、Play Web 框架应用程序、Grails 应用程序等。此外，它还支持插入许多著名的 Java 代理，例如 New Relic。

如果系统构建包不适用于你，并且你想使用一些不同的功能，那么你只需要使用 `cf push` 的 `-b` 参数，告诉 Cloud Foundry 去哪里查找构建包的代码，如示例 5-1 所示。

示例5-1 一个（非常）基本的Java EE servlet程序

```
cf push -b https://github.com/a/custom-buildpack.git#my-branch custom-app
```

或者，你可以在应用程序附带的 `manifest.yml` 文件中指定构建包。举例说明，假设我们有一个曾经通过 IBM 的 WebSphere 来部署的 Java EE 应用程序。IBM 需要维护一个非常庞大的 WebSphere Liberty 构建包。为了演示这一点，假设我们要部署和运行一个基本的 Servlet 程序（如示例 5-2 所示）。（现在我们暂时忽略，关于如何在 Spring Boot 应用程序中运行 Servlet 组件，可参考附录 A 中的介绍。）

示例5-2 一个（非常）基本的Java EE Servlet程序

```
package demo;
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```
@WebServlet("/hi")
```

```
public class DemoApplication extends HttpServlet {
```

```
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
```

```
        response.setContentType("text/html");
```

```
        response.getWriter().print("<html><body><h3>Hello Cloud</h3></body></html>");
```

```
    }
}
```


为了运行这个程序，我们在示例 5-3 中的应用程序清单中指定了 WebSphere Liberty 构建包。

示例5-3 WebSphere Liberty应用程序的manifest.yml文件

```
---
applications:
- name: wsl-demo
  memory: 1024M
  buildpack: https://github.com/cloudfoundry/ibm-websphere-liberty-buildpack.git
  instances: 1
  host: wsl-demo-${random-word}
  path: target/buildpacks.war
  env:
    SPRING_PROFILES_ACTIVE: cloud
    DEBUG: "true"
    debug: "true"
    IBM_JVM_LICENSE: L-JWOD-9SYNCP
    IBM_LIBERTY_LICENSE: L-MCAO-9SYMVC
```

一些构建包可以对自己本身进行自定义。Java 构建包 (<https://github.com/cloudfoundry/java-buildpack>) 最初是由 Heroku 开发的，而 Spring 和 Cloud Foundry 团队已经从根本上对它进行了扩展，能够支持通过环境变量进行配置。Java 构建包在 `config` 目录中提供了对其行为各个方面的默认配置。你可以通过指定一个与配置文件（以 `.yml` 扩展名结尾）同名的环境变量（前缀为 `JBP_CONFIG_`）来覆盖配置文件中所描述的行为。因此，借用一个优秀文档中 (<https://github.com/cloudfoundry/java-buildpack/blob/master/README.md>) 的例子，如果我们想要覆盖 JRE 版本和内存的配置（它们位于 `config/open_jdk_jre.yml` 文件中），可以使用示例 5-4 中的命令。

示例5-4 配置Java构建包的JDK和JRE

```
cf set-env custom-app JBP_CONFIG_OPEN_JDK_JRE \
'[jre: {version: 1.7.0_+}, memory_calculator: {memory_heuristics: {heap: 85,
stack: 10}}]'
```

容器化的应用程序

在 Java 世界中，我们为 J2EE / Java EE 应用程序服务器所开发的程序，往往对该应用程序服务器之外的迁移非常不友好。Java EE 应用程序为了夸大自己的可移植性，都会使用行为不一致的类加载器，提供不同的、带有专门配置文件的子系统，而且为了弥补很多可移植性方面的差异，它们经常提供服务器专有的 API。如果你的应用程序非常难以处理，并且已知的各种工具依然无法让你很有把握地进行迁移，我们依然还有一些希望！



在你不得不使用这个最后的手段之前，一定要先浏览社区中相关的构建包。例如，社区已经提供了基于 IBM 的 WebSphere（由 IBM 贡献，因为他们有一个基于 Cloud Foundry 的 PaaS 平台）和 RedHat 的 WildFly 的构建包。

Cloud Foundry 还支持运行容器化（例如 Docker）的应用程序（<http://bit.ly/2s6DSrl>）。如果你已经有了一个容器化的应用程序，并且只想用相同的工具链来部署和管理其他应用程序，那么这可能是一个替代方案。但是，由于以下各种原因，我们不建议采用这种方法：

- 因为 Docker 允许用户完全指定其根文件系统的内容，所以 Diego 上基于 Docker 的容器，其受到攻击的可能比使用构建包的可能性更高。相比之下，使用构建包的应用程序在受信任的根文件系统上运行。
- 构建一个容器，对开发人员和运维人员增加了额外的工作。
- 对于容器化的操作系统，管理补丁和更新会变得更加困难，因为需要重建它们才能看到更改结果。如果真的需要这样，现在没有简单的方法能够集中化地重建映像，以及重新部署每一个容器。

我们从常见的配置，介绍到应用程序和运行时特定的构建包，然后到容器式应用程序。我们会进行一些简化，然后开始尝试按照这个顺序对应用程序进行迁移。我们的目标是尽可能地减少我们的工作，尽可能的让 Cloud Foundry 替我们来完成。

将应用程序迁移到云上的微重构

在上一节中，我们介绍了一些方法，使用这些方法可以将应用程序整体从现有环境迁移到新的环境，而且不用修改代码。无论是需求常见的简单程序，还是极其复杂的应用程序，我们都介绍了相关的技术。可以看到，除了虚拟化应用程序并将其迁移至 Cloud Foundry 之外，还有其他的方法，但我们并没有考虑如何将应用程序指向它们所使用的后端服务（数据库、消息队列等）。为了简单起见，我们也忽略掉应用程序的某些类，只需要一些很小的、有时乏味但很简单的改动，就可以在 Cloud Foundry 上正常工作的情况。

在重构代码时，我们总希望有一个全面的测试套件来进行回归测试。但是我们也清楚，一些遗留的应用程序不会拥有这样的测试套件。可以谨慎行动，但是速度要快！

我们希望以最小的风险，通过一些微调让应用程序可以正常工作。毫无疑问，如果没有测试套件，更多的模块化代码将受到代码改动的影响。然而，非常讽刺的是，那些最需要全面测试套件的应用程序，恰恰是那些可能没有它们的应用程序：大型的、单体的、遗留的应用程序。如果你已经有了一个测试套件，你可能还缺少冒烟测试，来验证应用

程序及其相关服务的连接性和部署情况。这样一套测试虽然很难编写，但是在将遗留应用程序迁移到新环境中时，它们会非常有用。

连接后端服务

后端服务是应用程序使用的服务（数据库、消息队列、电子邮件服务等）。Cloud Foundry 应用程序通过在名为 `VCAP_SERVICES` 的环境变量中，查找后端服务的位置和凭据，来使用后端服务。这种方法的特点在于简单：任何语言都可以将环境变量从环境中提取出来，通过解析嵌入式的 JSON 来获取诸如服务主机、端口和凭证之类的信息。

如果应用程序依赖于由 Cloud Foundry 管理的后端服务，那么可以要求 Cloud Foundry 按需创建该服务。服务创建的过程也可以被称为配置（*provisioning*）。它的确切含义取决于上下文。对于电子邮件服务，它可能意味着提供新的电子邮件用户名和密码。对于 MongoDB 后端服务，它可能意味着创建一个新的 Mongo 数据库，并分配对该 MongoDB 实例的访问权限。后端服务的生命周期由 Cloud Foundry 的服务代理实例来管理。Cloud Foundry 服务代理是 Cloud Foundry 用来管理后台服务的 REST API。（第 14 章会更深入地探讨服务代理。）

一旦代理向 Cloud Foundry 注册了，那么你就可以通过 `cf market place` 命令，以及 `cf create-service` 命令来创建后端服务。我们来看一个假设的服务创建示例。在示例中，第一个参数 `mongo` 表示服务的名称。我们在这里使用的是通用名称，但它也可以是 New Relic、MongoHub、ElephantSQL 或 SendGrid 等。第二个参数是计划的名称，表示期望服务提供商所提供的服务级别和质量。有时候，服务级别越高意味着价格越高。第三个参数是前面提到的逻辑名称（如示例 5-5 所示）。

示例5-5 在Cloud Foundry上创建新的后端服务

```
cf create-service mongo free my-mongo
```

创建一个服务代理并不难，但可能需要做一些额外的工作。如果你希望应用程序与一个已有的、静态的、不太可能移动的服务进行交互，并且你只是想将应用程序指向它，则可以使用由用户提供的服务（*user-provided service*）（<http://bit.ly/2s6gPgp>）。“由用户提供的服务”是一个奇怪的说法，其实表示“给这个连接信息分配一个逻辑名称，让我能够像对待任何其他后端服务一样对待它，而实际上，它就是 `VCAP_SERVICES` 中的一条 JSON 项。

一个使用 `cf create-service` 命令创建的后端服务，可以作为一个由用户提供的服务，只有当它被绑定到一个应用程序上时，才对它的消费者可见。这会将相关的连接信息添加到该应用程序的 `VCAP_SERVICES` 配置中。

如果 Cloud Foundry 能够支持你需要的后端服务（例如 MySQL 或 MonodDB），并且你的代码已经支持对它们进行集中式的初始化或者获取（理想情况是通过依赖注入实现，这在 Spring 中非常简单），那么切换服务就是重新连接其他的依赖服务。如果你的应用程序已经支持十二要素风格的配置，其中密码、主机和端口等配置都在环境中进行维护，或者至少位于应用程序的外部，那么你不需要重新构建应用程序，就可以轻松地将其指向其新的服务。有关此内容更深入的介绍，请参阅第 3 章中有关十二要素应用程序风格的服务配置的讨论。

通常情况下，并没有这么简单。J2EE / Java EE 应用程序通常会查找大家熟知的上下文——JNDI（Java 命名和目录接口）来解析服务。如果你的代码中使用了依赖注入，那么非常简单，你只需要重新连接应用程序，就可以解析与 Cloud Foundry 环境的连接信息。如果没有解析，那么你需要修改代码，并且最好引入依赖注入来避免重复代码。如果在代码库中，只有一处代码（并且是唯一一处）指向 JNDI 或者 Cloud Foundry，那么你做得很好。组织良好的代码库应该能处理各种服务类型，例如 `javax.sql.DataSource` 和 `javax.jms.ConnectionFactory`，而不只是 JNDI 服务发现。

用 Spring 实现服务平等

在本节中，我们将讨论一些在将应用程序迁移到轻量级容器以及云计算环境中时，经常会遇到的困难。当然，这里只能列出实际中的一部分问题。

远程过程调用

Cloud Foundry（包括实际上的大部分云计算平台）会优先使用 HTTP。它支持单独的可寻址节点，甚至支持不可路由的自定义端口，但是这些功能并不完美，并且不是所有环境都支持。例如，如果你使用 RMI / EJB 进行 RPC 调用，那么就必须通过 HTTP 进行隧道传输。这里为了更好地理解基于 HTTP 的 RPC，我们暂时忽略使用 RPC 的细节。有很多方法可以实现这一点，包括 XML-RPC、SOAP，甚至 Spring 的 HTTP Invoker 服务导出器（<http://bit.ly/2s6qYti>）和服务客户端，它们都可以通过 HTTP 来完成 RMI 通信传输。其中最后一个方法在使用上很方便，因为它能够通过 Java 序列化支持 rich API 和 Java 类型，但是通过 HTTP 来传输数据。不过，这样做会将客户端和生产者耦合在一起，并且必须在类型使用上达成一致，但这不是我们讨论的方向。示例 5-6 演示了如何使用 Spring 的 HTTP Invoker，通过 `SimpleMessageService` 的接口来导出一个 Spring bean。

示例5-6 使用HttpInvokerServiceExporter导出服务

package demo;

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter;
```

@SpringBootApplication

public class DemoApplication {

```
    public static void main(String[] args) throws Exception {
        SpringApplication.run(DemoApplication.class, args);
    }
```

@Bean

```
    MessageService messageService() { ❶
        return new SimpleMessageService();
    }
```

❷

@Bean(name = "/messageService")

```
    HttpInvokerServiceExporter httpMessageService() {
        HttpInvokerServiceExporter http = new HttpInvokerServiceExporter();
        http.setServiceInterface(MessageService.class);
        http.setService(this.messageService());
        return http;
    }
}
```

- ❶ 代码本身至少需要实现 HttpInvokerServiceExporter 中指定的服务接口。
- ❷ HttpInvokerServiceExporter 将指定的 bean 映射到 Spring DispatcherServlet 下的 HTTP 端点 (/messageService)。

为了支持序列化, Message 需要实现 java.io.Serializable 接口, 就像使用直接的 RMI 序列化一样。Spring 提供了镜像组件, 可以为这些使用同一接口的远程服务创建客户端。在示例 5-7 中, 我们将使用 HttpInvokerProxyFactoryBean 为远程服务创建一个客户端代理, 将其绑定到相同的服务合约上。但是, 这种基于共享合约的 RPC 有个缺陷, 就是它将客户端与服务类型耦合了起来, 使得服务无法在不影响客户端的情况下发生改变。



有一些新的 RPC 框架, 例如 gRPC, 可以支持对传输数据的临时修改。在这些技术中, 只要有一个共同的子集, 客户端和服务就可以进行交互。

我们可以很轻松地使用 HttpInvokerProxyFactoryBean 来建立一个与该端点交互的客户

端。这是 `HttpInvokerServiceExporter` 的另一种实现方式。

示例5-7 在 `DemoApplicationTests.java` 中创建并调用一个RPC客户端

```
package demo;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean;
```

```
@Configuration
```

```
public class DemoApplicationClientConfiguration {
```

```
    @Bean
```

```
    HttpInvokerProxyFactoryBean client() {
```

```
        HttpInvokerProxyFactoryBean client = new HttpInvokerProxyFactoryBean();
```

```
        client.setServiceUrl("http://localhost:8080/messageService"); ❶
```

```
        client.setServiceInterface(MessageService.class); ❷
```

```
        return client;
```

```
    }
```

```
}
```

❶ 这里的 URL 是在服务 bean 上指定的字符串名称。

❷ 客户端代码应该注入 `messageService` 类型，以便能够调用下游的服务。

使用 Spring Session 的 HTTP 会话

Cloud Foundry（以及大多数的云计算环境）在多播网络方面效果不佳。HTTP 会话复制通常与多播网络有关。在传统的应用服务器中，会话复制指的是在单个集群上进行网络广播。不幸的是，这些传统的会话复制方案并不具有良好的性能、健壮性或者可移植性。

Spring Session (<http://spring.io/projects/spring-session>) 可以在这方面提供帮助。Spring Session 依赖于 SPI 来处理会话同步，可以完全替代 Servlet HTTP Session API。SPI 的实现可以和所有的后端进行交互，包括 Redis、Apache Geode 和 Hazelcast。这些项目的重点是如何实现集群，以及实现跨集群复制的可靠性。另外，它们很容易在大多数云平台上得到支持。例如，Redis 在大多数的 Cloud Foundry 安装版本上都会预部署，并且还会提供一个方便的 Hazelcast 服务代理 (<https://docs.pivotal.io/partners/hazelcast/>)。



像 Redis 和 Hazelcast 这样的技术很重要，但是它们会增加运维成本。理想情况下，这个能力应该由平台来自动管理。假设有两个大致相同的功能，那我们应选择一个最容易运维的。

你只需安装 Spring Session 即可使用 HTTP 会话。HTTP servlet 规范以这种方式提

供替换实现方式，所以它们都可以在 `servlet` 的实现中工作。随后，`Spring Session` 会通过 `SPI` 写入会话状态。`Redis` 可作为 `Cloud Foundry` 的后台服务。当多个节点启动后，它们都会与同一个 `Redis` 集群进行通信，并得益于 `Redis` 著名的状态复制机制。为了使 `Redis` 和 `Spring Session` 能够正常工作，我们需要在 `Spring Boot` 应用程序中添加 `org.springframework.boot:spring-boot-starter-redis` 和 `org.springframework.session:spring-session` 依赖。



`Servlet API` 要求任何写入 `HTTP` 会话的对象都必须支持 `Java` 序列化。对于 `Spring Session` 实现来说，可能也需如此！但是从另一方面来说，这也不是必须的，因为有些后端服务可以不依靠 `Java` 序列化来完成工作。

`Spring Session` 还为你免费提供了一些其他的功能：

- 支持使用 `WebSockets` 的 `HTTP` 会话。
- 支持简单的“注销我的账户”类的功能。
- 支持访问逻辑上不同的会话。也就是说，两个不同的应用程序也可以共享相同的会话状态。

有关更多信息，请参阅 `Spring` 博客上的“可移植的、为云准备的 `HTTP` 会话”(<http://bit.ly/2s6CRzE>) 文章。

如果你想运行示例 5-8，可以使用 `Redis CLI` 启动 `Redis`，然后使用 `FLUSH ALL` 命令将其清空。

示例5-8 `DemoApplication.java`

```
package demo;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import javax.servlet.http.HttpSession;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;
import java.util.UUID;
```

```
@SpringBootApplication
public class DemoApplication {
```

```

public static void main(String[] args) {
    SpringApplication.run(DemoApplication.class, args);
}
}

@RestController
class SessionController {

    private final String ip;

    @Autowired
    public SessionController(@Value("${CF_INSTANCE_IP:127.0.0.1}") String ip) {
        this.ip = ip;
    }

    @GetMapping("/hi")
    Map<String, String> uid(HttpSession session) {
        ❶
        UUID uid = Optional.ofNullable(UUID.class.cast(session.getAttribute("uid")))
            .orElse(UUID.randomUUID());
        session.setAttribute("uid", uid);

        Map<String, String> m = new HashMap<>();
        m.put("instance_ip", this.ip);
        m.put("uuid", uid.toString());
        return m;
    }
}

```

- ❶ 在该示例中，如果 HTTP 会话不存在某个属性，会将它存储在会话中。后续对 /hi 端点的调用，应该会返回与 HTTP 会话中相同的缓存值。



示例 5-9 将删除 Redis 数据库中的所有内容!

示例5-9 一个空的Redis数据库

```
127.0.0.1:6379> flushall
OK
```

```
127.0.0.1:6379> keys *
(empty list or set)
```

然后，打开 Web 应用程序的 `http://localhost:8080/hi` 地址。刷新几次，你会发现 `uuid` 中的值在第一次刷新之后就不变了，这说明第一次触发了一个新的 HTTP 会话，并且 Spring 会将它保存在 Redis 中。我们再次在 Redis 实例上运行 `KEYS *` 命令来确认结果，

如示例 5-10 所示。

示例5-10 含有一些会话数据的Redis数据库

```
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> keys *
1) "spring:session:expirations:1490589000000"
2) "spring:session:sessions:expires:7c82002e-dd4d-4196-b9ae-1d93037192f4"
3) "spring:session:sessions:7c82002e-dd4d-4196-b9ae-1d93037192f4"
```

如果再次运行 FLUSHALL 命令，它将重置数据库，并且下一次浏览器刷新将产生一个新的 uuid 值。

Java 消息服务

Cloud Foundry 没有提供好的 JMS 解决方案。按照 RabbitMQ 的说法，大部分 JMS 代码应该换成使用 AMQP 协议。如果你使用的是 Spring，那么处理 JMS 或 RabbitMQ（或者 Redis 的发布订阅功能）的代码都是相似的。RabbitMQ 和 Redis 都可在 Cloud Foundry 上使用。另一种方法是使用 RabbitMQ JMS 客户端 (<https://github.com/rabbitmq/rabbitmq-jms-client>)。

通过 X/Open XA 协议和 JTA 实现分布式事务

如果应用程序需要使用 XA/Open 协议和 JTA 来支持分布式事务，那么可以使用 Spring 来配置单独的 XA 提供器 (<http://bit.ly/2s6fg1Q>)，使用 Spring Boot (<http://bit.ly/2s6mXFw>) 会更加容易。你不需要一个由 Java EE 容器管理的 XA 事务管理器。示例 5-11 定义了一个 JMS 消息侦听器和一个基于 JPA 的服务。

示例5-11 通过Spring Boot的JTA自动配置来使用JTA

package demo;

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
```

```

import javax.transaction.Transactional;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(DemoApplication.class, args);
    }
}

interface AccountRepository extends JpaRepository<Account, Long> {
}

@Entity
class Account {

    @Id
    @GeneratedValue
    private Long id;

    private String username;

    Account() {
    }

    public Account(String username) {
        this.username = username;
    }

    public String getUsername() {
        return this.username;
    }
}

@Component
class Messages {

    private Log log = LoggerFactory.getLogger(getClass());

    @JmsListener(destination = "accounts")
    public void onMessage(String content) {
        log.info("----> " + content);
    }
}

@Service
@Transactional
class AccountService {

    @Autowired
    private JmsTemplate jmsTemplate;

    @Autowired

```



```

private AccountRepository accountRepository;

public void createAccountAndNotify(String username) {
    this.jmsTemplate.convertAndSend("accounts", username);
    this.accountRepository.save(new Account(username));
    if ("error".equals(username)) {
        throw new RuntimeException("Simulated error");
    }
}
}

```

Spring Boot 会在全局事务中自动使用 JDBC XADataSource 和 JMS XAConnectionFactory 资源。我们通过一个单元测试来演示，无论是 JDBC DataSource 还是 JMS ConnectionFactory，触发回滚后对数据都没有影响（如示例 5-12 所示）。

示例5-12 JTA演示

```

package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import static org.junit.Assert.assertEquals;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = DemoApplication.class)
public class DemoApplicationTests {

    private Log log = LogFactory.getLog(getClass());

    @Autowired
    private AccountService service;

    @Autowired
    private AccountRepository repository;

    @Test
    public void contextLoads() {
        service.createAccountAndNotify("josh");
        log.info("count is " + repository.count());
        try {
            service.createAccountAndNotify("error");
        }
        catch (Exception ex) {
            log.error(ex.getMessage());
        }
    }
}

```

```

log.info("count is " + repository.count());
assertEquals(repository.count(), 1);
}
}

```

Spring Boot 允许指定多个属性，来配置底层的 JTA 实现（Bitronix, Atomikos）存储事务日志的位置。



理想情况下，事务日志应该存储在一个持久的地方。Cloud Foundry 上的应用程序没有可以确保安全的文件系统。如果应用程序的实例出现故障，或者实例重新启动（也可能在另外的节点上启动），我们无法保证文件系统的内容在其他节点上可用。需要使用一些更持久化的手段，来保证数据的绝对可恢复。

云文件系统

Cloud Foundry 没有提供一个可持久化的文件系统。你可以在 Cloud Foundry 上使用基于 FUSE 的文件系统，例如 SSHFS。FUSE 是一个在用户空间中构建文件系统实现的 C / C++ API。有各种基于 FUSE 的文件系统，可以将 HTTP API、SSH 连接、MongoDB 文件系统等来源，以文件系统的形式安装到类 UNIX 操作系统中。例如，你可以在用户空间中挂载一个使用 SSH 的远程文件系统。当然，这样的话，你需要一个可以通过 SSH 访问的远程机器。该方案不仅实际有效，并且在使用 JTA 时非常方便，你只需要通过 `java.io.File` 来保证数据的完整性。但是，这种方案的访问速度会比较慢。

如果你只需要读取和写入字节，而不关心是否通过 `java.io.File` 或者类文件系统的后端服务来操作 I / O，那么你可以考虑其他更合适的备选方案，例如基于 MongoDB GridFS 的解决方案 (<http://bit.ly/2s6d8re>) 或基于 Amazon Web 服务的 S3 解决方案。（我们会在第 14 章中讨论创建服务代理时，介绍如何连接到 Amazon S3）。这些后端服务都提供了类似于文件系统的 API。你可以通过一个逻辑名称读取、写入和查询多个字节。Spring Data MongoDB 提供了非常方便的 `GridFsTemplate` 类，使用它可以简化读写数据的工作。

我们来看一个将文件上传到 MongoDB 后端的例子。当你运行应用程序时，可以打开 <http://localhost:8080>，与 REST API 进行交互。



像 MongoDB 这样的技术是非常重要的，但是它们会增加运维工作量。理想情况下，这个功能应该由平台自动管理。如果你使用的是 Cloud Foundry，那么在服务目录中有一个 MongoDB 服务。

示例5-13 使用MongoDB的GridFS，作为一个读取和写入文件数据的REST API的后端服务

```
package demo;

import com.mongodb.gridfs.GridFSDBFile;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.data.mongodb.gridfs.GridFsCriteria;
import org.springframework.data.mongodb.gridfs.GridFsTemplate;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.io.ByteArrayOutputStream;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}

@Controller
@RequestMapping(value = "/files")
class FileController {

    @Autowired
    private GridFsTemplate gridFsTemplate;

    private static Query getFilenameQuery(String name) {
        return Query.query(GridFsCriteria.whereFilename().is(name));
    }

    ❶
    @RequestMapping(method = RequestMethod.POST)
    String createOrUpdate(@RequestParam MultipartFile file) throws Exception {
        String name = file.getOriginalFilename();
        maybeLoadFile(name).ifPresent(
            p -> gridFsTemplate.delete(getFilenameQuery(name)));
        gridFsTemplate.store(file.getInputStream(), name, file.getContentType())
            .save();
        return "redirect:/";
    }
}
```

```

}

②
@RequestMapping(method = RequestMethod.GET)
@ResponseBody
List<String> list() {
    return getFiles().stream().map(GridFSDBFile::getFilename)
        .collect(Collectors.toList());
}

③
@RequestMapping(value = "/{name:.+}", method = RequestMethod.GET)
ResponseBody<?> get(@PathVariable String name) throws Exception {
    Optional<GridFSDBFile> optionalCreated = maybeLoadFile(name);
    if (optionalCreated.isPresent()) {
        GridFSDBFile created = optionalCreated.get();
        try (ByteArrayOutputStream os = new ByteArrayOutputStream()) {
            created.writeTo(os);

            HttpHeaders headers = new HttpHeaders();
            headers.add(HttpHeaders.CONTENT_TYPE, created.getContentType());
            return new ResponseEntity<byte[]>(os.toByteArray(), headers, HttpStatus.OK);
        }
    }
    else {
        return ResponseEntity.notFound().build();
    }
}

private List<GridFSDBFile> getFiles() {
    return gridFsTemplate.find(null);
}

private Optional<GridFSDBFile> maybeLoadFile(String name) {
    GridFSDBFile file = gridFsTemplate.findOne(getFilenameQuery(name));
    return Optional.ofNullable(file);
}
}

```

① /files 端点可以接受上传的文件数据，并将其写入 MongoDB 的 GridFS。

② /files 端点只是返回一个 GridFS 中的文件列表。

③ /files/{name} 端点从 GridFS 中读取多个字节，并将它们发送回客户端。

另外，如果你的应用程序只是临时需要使用文件系统，那么你可以使用 Cloud Foundry 应用程序的临时目录，但请记住，Cloud Foundry 不保证数据能够保存多久。

HTTPS

为了保护所有应用程序，Cloud Foundry 会在一个高可用代理中终止所有的 HTTPS 请求。

所有路由到应用程序的请求也会要求使用 HTTPS。如果你使用的是私有化的 Cloud Foundry，可以提供自己的证书。

电子邮件

你的应用程序是否使用 SMTP / POP3 或 IMAP？如果你在 Java 应用程序中使用了电子邮件，那么很可能使用的是 JavaMail。JavaMail 是一个支持 SMTP / POP3 / IMAP 的电子邮件 API。市面上有许多提供电子邮件服务的供应商。SendGrid (<http://bit.ly/2s6yNPK>) 就是一个 Spring Boot 原生支持的电子邮件云服务商，提供简单的邮件 API。我们来看一个例子。



像 SendGrid 这样的技术很适合云原生应用程序，因为它们本身就是在云上托管的。你甚至可以使用 Cloud Foundry 上的某些服务目录（例如 Pivotal Web Services）来创建与这些服务的绑定。

示例5-14 Spring Boot对SendGrid的支持

```
package demo;

import com.sendgrid.SendGrid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}

@RestController
class EmailRestController {

    @Autowired
    private SendGrid sendGrid;

    1 @RequestMapping("/email")
    SendGrid.Response email(@RequestParam String message) throws Exception {
        SendGrid.Email email = new SendGrid.Email();
        email.setHtml("<h1>" + message + "</h1>");
        email.setText(message);
    }
}
```

```

email.setTo(new String[] { "user1@host.io" });
email.setToName(new String[] { "Josh" });
email.setFrom("user2@host.io");
email.setFromName("Josh (sender)");
email.setSubject("I just called.. to say.. I (message truncated)");
return sendGrid.send(email);
}
}

```

- ❶ 该 REST API 将一条消息作为请求参数，使用自动配置的 SendGrid Java 客户端来发送电子邮件。自动配置需要一个有效的 SendGrid 用户名 (`spring.sendgrid.username`) 和密码 (`spring.sendgrid.password`)。请记住，Spring Boot 会规范化属性的名称，因此你也可以通过环境变量来设置这些值，例如 `SPRING_SENDGRID_USERNAME`、`SPRING_SENDGRID_PASSWORD` 等。

身份管理

身份管理、身份认证和授权是分布式系统中非常重要的功能。集中式描述用户、角色和权限的能力至关重要。Cloud Foundry 提供了一个强大的身份认证和授权服务 UAA，可以通过 Spring Security 来调用。另外，你可能会发现 Stormpath (<https://stormpath.com>) 也是一个不错的第三方托管服务，可以作为其他身份服务提供方或者它自己的一个代理。它甚至提供了一个非常简单的 Spring Boot 和 Spring Security 集成 SDK (<https://github.com/stormpath/stormpath-sdk-java>)！



像 Okta 这样的技术非常重要，因为它们本身是在云上托管的。相比 Active Directory 这种可能需要一个全职管理员的方案，它极大地减少了运维的开销！



在 2017 年初，Okta 收购了 Stormpath，承诺保留原来 Stormpath 应用程序的体验，但是将它们连接到 Okta 的后端服务。

总结

希望你在本章中找到一些东西，来帮助你迁移一个棘手的应用程序！如果你打算这么做，那么就需要考虑编写测试来自动验证你的期望。如果你预计在迁移数据库访问、文件系统访问和电子邮件时会遇到问题，那么可以设置一个连接到数据库的端点、一个发送电子邮件的端点以及一个以需要的方式读取文件的端点。在你迁移应用程序之前，应

该使用持续集成技术来部署应用程序并测试这些端点。一旦所有端点都正常工作，并且确信其他功能也都正常，那么就进行一次更彻底的测试，最终验收应用程序。如果你担心在应用程序中留下这些端点，那么尽管可以先删除它们，但是一定要添加一个详细的TODO 或者待办项，以便以后将它们添加到 Spring Boot Actuator 中。Actuator 提供了一个很容易被保护起来的 /health 端点，其也能够获得同类信息。在处理待办任务的时候，请优先考虑编写适当的测试！

本章的目标是解决将现有遗留应用程序迁移到云时会遇到的一些常见问题。通常在迁移中，都会遇到我们这里提到的几个问题。一旦你完成了迁移，应该好好庆祝一下。迁移之后，你就不需要再管理和担心这些工作了，云平台会来接管它们。

第 II 部分

Web服务

REST API

曾经有一段时间，因为系统需要共享访问多个不同团队的数据库，亚马逊很难扩展由不同团队负责的功能。这使得单个团队，很难在不影响公司其他团队的情况下，独自去更改系统的功能。所以，正如亚马逊公司首席技术官 Werner Vogels 在 2006 年所解释的 (<http://bit.ly/2s6wwEr>)，所有的集成都应该通过 API 而不是数据库来实现。

这是迈向微服务重要的第一步：一切都是 API。Representational State Transfer (REST) 是网络上数百万个 API 中最流行的协议。

REST 最初是由 Roy Fielding 博士在 2000 年作为其博士论文 (<http://bit.ly/2j4SIKI>) 的一部分提出的。Fielding 曾经帮助定义了 HTTP 规范，并且受邀帮助说明如何使用已经被证明的、可以大规模扩展的、去中心化的、抗失败的 Web 结构来构建服务。HTTP 的存在恰好是 REST 架构优点的证明。

在以前构建分布式服务（比如 CORBA、EJB、RMI 和 SOAP）的方法中，从某种程度上都侧重于，如何将一个面向对象的接口和方法，暴露成可以远程访问的服务 (RPC)。相反，REST 则侧重于对远程资源或实体的操作。注意，这里的操作是名词，而不是动词，REST 关注的是实体，而不是行为。

伦纳德·理查森的成熟模型

REST 的核心，就是使用 HTTP 已经为服务间通信所提供的动词 (GET、PUT、POST、DELETE 等) 和习惯用语 (HTTP 标头、状态码等)，来描述业务状态的变化。最好的 REST API 倾向于利用更多 HTTP 的功能。REST 带来的所有好处，其实都是一种对 HTTP 的架构约束，而不是某种标准，所以我们看到出现了大量各种在不同程度上符合 RESTful 原则的 API。为此，伦纳德·理查森 (Leonard Richardson) 提出了他的 REST

成熟度模型 (<http://bit.ly/2s6elP9>), 来帮助评估一个 API 符合 REST 原则的程度。

0 级：使用 POX (Plain Old XML)

当然, 等级应当从 0 开始! 该等级描述了使用 HTTP 作为传输协议的 API, 仅此而已。例如, 基于 SOAP 的 Web 服务使用了 HTTP, 但它们也可以使用 JMS, 因此它们只是偶尔基于 HTTP 的。这样的服务主要将 HTTP 作为一个访问 URI 的隧道。SOAP 和 XML-RPC 就是例子。它们通常只使用一个 HTTP 动词 (POST)。

1 级：资源

描述使用多个 URI 来区分相关名词的 API, 但是没有使用 HTTP 的全部功能。

2 级：HTTP 动词

描述利用传输本身的属性 (例如 HTTP 动词和状态代码) 来增强服务的 API。即使你错误地使用了 Spring MVC、JAX-RS 或者任何其他 REST 框架, 那么最终的 API 仍然可能符合 2 级标准! 这是一个很好的起点。

3 级：超媒体控件 (HATEOAS, 表示作为应用程序状态引擎的超媒体)

描述不需要预知服务内容即可访问的 API。这种服务会通过一个统一形式来展示服务的某个结构, 从而提升该服务的使用范围。

在本章中, 我们将介绍如何使用 Spring 从第 2 级开始来构建 REST 服务。稍后, 将介绍如何使用超媒体来实现统一的自描述服务。REST API 用来表示系统服务之间的 HTTP 约定。虽然使用这些约定的是 API 客户端, 但是开发者必须针对它们进行开发, 所以我们会特别关注如何开发易于理解的、正确的和文档一致的 API。



本章我们将使用 Spring MVC。Spring 也可以和 JAX-RS 一起使用。如果你使用 Spring Initializr (<http://start.spring.io>), 则为需要支持 JAX-RS 的 Web 应用程序选择 “Jersey (JAX-RS)”。此外, 使用 Spring 的其他实现也不难。有关集成其他 servlet 框架的更多信息, 请参阅附录 A 中关于如何在 Spring Boot 应用程序中使用传统 Java EE API (例如 servlet) 的讨论。

REST 是云原生应用程序的一个重要组成部分。虽然微服务没有强制或者要求使用 REST, 但它是暴露服务最普遍的方法。

HTTP 非常适用于云原生应用程序。HTTP 是服务缓存的理想选择, 因为它没有客户端状态。相反, HTTP 的每个请求都是独立的。这也意味着只要 REST API 所表示的状态可以水平扩展, 那么服务也就可以轻松地被水平扩展。缓存缩短了发送第一个字节的时间。通过使用缓存和 GZip 压缩, 你可以大幅提升单个 HTTP 请求的性能, 如果再配合

使用 HTTP 2，那么会在短期内看到非常显著的效果。

HTTP 的内容类型是不可预知的，因此它支持内容协商。一个客户端可以支持 XML、JSON 或者 Google 的 Protocol Buffers 其中之一。同一个服务可以支持所有的协议。这个机制也是可扩展的。

使用 Spring MVC 实现简单的 REST API

为了在 Spring MVC 中引入一个 servlet 容器和完整的配置，将 `org.springframework.boot:spring-boot-starter-web` 添加到你的构建文件中。Spring MVC 是一个面向请求/响应的 HTTP 框架。在 Spring MVC 中，控制器中的处理器方法会被映射成 HTTP 请求，并最终提供响应。我们来看一个用来操作 Customer 实体的 REST API(如示例 6-1 所示)。

示例6-1 我们的第一个@RestController, CustomerRestController

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.method.annotation.MvcUriComponentsBuilder;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.net.URI;
import java.util.Collection;

@RestController
@RequestMapping("/v1/customers")
public class CustomerRestController {

    @Autowired
    private CustomerRepository customerRepository;

    ❶
    @RequestMapping(method = RequestMethod.OPTIONS)
    ResponseEntity<?> options() {

        // @formatter:off
        return ResponseEntity
            .ok()
```

```

        .allow(HttpMethod.GET, HttpMethod.POST,
            HttpMethod.HEAD, HttpMethod.OPTIONS,
            HttpMethod.PUT, HttpMethod.DELETE)
        .build();
    // @formatter:on
}

@GetMapping
ResponseBody<Collection<Customer>> getCollection() {
    return ResponseEntity.ok(this.customerRepository.findAll());
}

2
@GetMapping(value =("/{id}")
ResponseBody<Customer> get(@PathVariable Long id) {
    return this.customerRepository.findById(id).map(ResponseEntity::ok)
        .orElseThrow(() -> new CustomerNotFoundException(id));
}

@PostMapping
ResponseBody<Customer> post(@RequestBody Customer c) { 3

    Customer customer = this.customerRepository.save(new Customer(c
        .getFirstName(), c.getLastName()));

    URI uri = MvcUriComponentsBuilder.fromController(getClass()).path("/{id}")
        .buildAndExpand(customer.getId()).toUri();
    return ResponseEntity.created(uri).body(customer);
}

4
@DeleteMapping(value =("/{id}")
ResponseBody<?> delete(@PathVariable Long id) {
    return this.customerRepository.findById(id).map(c -> {
        customerRepository.delete(c);
        return ResponseEntity.noContent().build();
    }).orElseThrow(() -> new CustomerNotFoundException(id));
}

5
@RequestMapping(value =("/{id}", method = RequestMethod.HEAD)
ResponseBody<?> head(@PathVariable Long id) {
    return this.customerRepository.findById(id)
        .map(exists -> ResponseEntity.noContent().build())
        .orElseThrow(() -> new CustomerNotFoundException(id));
}

6
@PutMapping(value =("/{id}")
ResponseBody<Customer> put(@PathVariable Long id, @RequestBody Customer c) {
    return this.customerRepository
        .findById(id)

```



```

.map(
existing -> {
    Customer customer = this.customerRepository.save(new Customer(existing
        .getId(), c.getFirstName(), c.getLastName()));
    URI selfLink = URI.create(ServletUriComponentsBuilder.fromCurrentRequest()
        .toUriString());
    return ResponseEntity.created(selfLink).body(customer);
}).orElseThrow(() -> new CustomerNotFoundException(id));
}
}

```

- ❶ 处理器方法由 `@RequestMapping` 注解指定。一个类可能包含一个 `@RequestMapping` 注解，在这种情况下，方法级声明的映射会覆盖或添加到类级别的映射中。如果客户端想知道给定资源支持哪些 HTTP 动词，它可以发出动作为 `OPTIONS` 的 HTTP 请求，再由处理器方法负责响应。
- ❷ 该处理器方法通过 URI 语法 `{id}`，将 ID 编码到 `@RequestMapping` 的路径变量中，再返回由 ID 指定的记录。
- ❸ 数据可能会从客户端传输到服务端，并且数据内容会作为处理器的 `@RequestBody` 参数传递给服务。
- ❹ `DELETE` 处理器在成功时会返回 HTTP 204，或者在失败时抛出一个异常，返回 404 状态码。
- ❺ `HEAD` 处理器只是为了确认资源的存在，所以它在成功时会返回 204，或者在失败时会抛出异常并返回 404，就像在 `DELETE` 处理器中一样。
- ❻ `PUT` 处理器负责更新现有的记录。它使用 `@PathVariable` 参数来指定要更新的记录。如果记录不存在，则抛出一个 404 异常。

这是一个基础的 Spring MVC REST 控制器。`@RequestMapping` 注解会将处理器方法映射到 HTTP 请求类型上。`@RequestMapping` 注解让我们可以进一步根据请求中的标题、发送和返回的内容类型以及 Cookie 等进行区分。这里处理器指定的路径，是相对于应用程序的上下文根路径而言的，除非我们在控制器类的 `@RequestMapping` 注解中指定一个其他的路径。



Spring MVC 也支持像 `@GetMapping`、`@PostMapping` 等在方法上使用的 HTTP 注解。它们的作用跟 `@RequestMapping` 一样，只不过只能适用于 HTTP 方法。如果你的端点不需要支持多个 HTTP 方法，那么你也可以使用这些注解。



内容协商

处理器方法会返回一个对象，或者包含对象的 `ResponseEntity` 封装。当 Spring MVC 遇到一个返回值时，它会使用一个策略对象 `HttpMessageConverter`，将方法返回对象转换成适合客户端的表现形式。客户端通过内容协商来指定期望的表现形式。默认情况下，Spring MVC 会在请求的 `Accept` 头中，查找 `HttpMessageConverter` 支持创建的某种媒体类型，例如 `application/json`、`application/xml` 等，将方法返回对象转换成对应的格式。对于从客户端向服务发送的数据，会按照相反的顺序执行相同的流程，并以 `@RequestBody` 参数的形式将数据传递给处理器方法。

内容协商是 HTTP 最强大的功能之一：同一个服务可以同时支持多个使用不同协议的客户端。我们可以使用媒体类型来告诉客户端，当前所提供的内容类型。通常，媒体类型也代表客户端如何处理响应中的内容。例如，客户端知道不要试图从媒体类型为 `image/jpeg` 的响应中，提取 JSON 字符串。

读写二进制数据

到目前为止，我们已经熟悉了 JSON，但是 REST 资源并非不能提供像图像或者文件这样的媒体类型。现在我们就来看一看，如何在个人档案接口 `/customers/{id}/photo` 读取和写入图像数据（如示例 6-2 所示）。

示例6-2 读取和写入二进制（图像）数据

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.Resource;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.util.Assert;
import org.springframework.util.FileCopyUtils;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import java.io.*;
import java.net.URI;
import java.util.concurrent.Callable;

//@formatter:off
import static org.springframework.web.servlet
    .support.ServletUriComponentsBuilder.fromCurrentRequest;
```




```
//@formatter:on
```

```
@RestController
```

```
@RequestMapping(value = "/customers/{id}/photo")
```

```
public class CustomerProfilePhotoRestController {
```

```
    private File root;
```

```
    private final CustomerRepository customerRepository;
```

```
    private final Log log = LoggerFactory.getLog(getClass());
```

```
    @Autowired
```

```
    CustomerProfilePhotoRestController(
```

```
        @Value("${upload.dir:${user.home}/images}") String uploadDir,
```

```
        CustomerRepository customerRepository) {
```

```
        this.root = new File(uploadDir);
```

```
        this.customerRepository = customerRepository;
```

```
        Assert.isTrue(this.root.exists() || this.root.mkdirs(),
```

```
            String.format("The path '%s' must exist.", this.root.getAbsolutePath()));
```

```
    }
```

❶

```
@GetMapping
```

```
ResponseBody<Resource> read(@PathVariable Long id) {
```

```
    return this.customerRepository
```

```
        .findById(id)
```

```
        .map(
```

```
            customer -> {
```

```
                File file = fileFor(customer);
```

```
                Assert.isTrue(file.exists(),
```

```
                    String.format("file-not-found %s", file.getAbsolutePath()));
```

```
                Resource fileSystemResource = new FileSystemResource(file);
```

```
                return ResponseEntity.ok().contentType(MediaType.IMAGE_JPEG)
```

```
                    .body(fileSystemResource);
```

```
            }).orElseThrow(() -> new CustomerNotFoundException(id));
```

```
    }
```

❷

```
@RequestMapping(method = { RequestMethod.POST, RequestMethod.PUT })
```

```
Callable<ResponseBody<?>> write(@PathVariable Long id,
```

```
    @RequestParam MultipartFile file) ❸
```

```
    throws Exception {
```

```
    log.info(String.format("upload-start /customers/%s/photo (%s bytes)", id,
```

```
        file.getSize()));
```

```
    return () -> this.customerRepository
```

```
        .findById(id)
```

```
        .map(
```

```
            customer -> {
```

```
                File fileForCustomer = fileFor(customer);
```

```
                try (InputStream in = file.getInputStream();
```

```
                    OutputStream out = new FileOutputStream(fileForCustomer)) {
```

```
                    FileCopyUtils.copy(in, out);
```



```

    }
    catch (IOException ex) {
        throw new RuntimeException(ex);
    }
    URI location = fromCurrentRequest().buildAndExpand(id).toUri(); ❹
    log.info(String.format("upload-finish /customers/%s/photo (%s)", id,
        location));
    return ResponseEntity.created(location).build();
}).orElseThrow(() -> new CustomerNotFoundException(id));
}

private File fileFor(Customer person) {
    return new File(this.root, Long.toString(person.getId()));
}
}

```

- ❶ Spring MVC 会从 Spring MVC 控制器方法返回的 `org.springframework.core.io.Resource` 底层缓冲区中，自动返回 `byte []` 类型的数据，例如文件、URL 资源、`InputStream` 等。
- ❷ 文件上传可能会阻塞和独占 Servlet 容器的线程池。Spring MVC 后台的 `Callable<T>` 处理器方法会将值返回给配置好的 `Executor` 线程池，并释放容器线程，直到响应准备就绪。
- ❸ Spring MVC 会自动将上传的文件分块数据，映射到一个 `org.springframework.web.multipart.MultipartFile` 类型的请求参数上。
- ❹ 建议你在创建一个资源后，返回一个 HTTP 201（已创建）状态代码，并返回一个 `Location` 头信息，指向新创建的资源 URI。

如果媒体数据可以用 Spring 的 `org.springframework.core.io.Resource` 实现来表示，那么读取媒体内容就会变得很容易。有很多现成的实现可供选择：`FileSystemResource`、`GridFsResource`（基于 MongoDB 的网格文件系统）、`ClassPathResource`、`GzipResource`、`VfsResource`、`UrlResource` 等。Spring MVC 会自动从 `org.springframework.core.io.Resource` 底层的缓存中，将 `byte[]` 数据返回给客户端。

Spring MVC 应用程序运行在一个 servlet 容器中。servlet 容器本身会维护一个前端的线程池，用来响应接收到的 HTTP 请求。每当接收到一个 HTTP 请求，容器就会分发一个线程来处理请求并生成回复响应。重要的是，不要耗尽 servlet 容器线程池中的线程。如果控制器的处理器方法返回一个 `java.util.concurrent.Callable<T>`、`org.springframework.web.context.request.async.DeferredResult` 或者 `org.springframework.web.context.request.async.WebAsyncTask` 对象，那么 Spring MVC 会将该方法移到一个单独的线程池（通过一个 `TaskExecutor` bean 来指定）中执行。





Spring MVC 还支持 websockets 和 Server-Sent Events (SSE)，它们都是异步的通信机制，但是各有不同，我们这里不会对它们进行讨论。

`Callable<T>` 只是一个特殊的、返回结果的 `Runnable` 实例。Spring MVC 会调用指定线程池上的 `Callable` 任务，一旦它产生了一个结果，就可以将结果异步地返回给原始的 HTTP 请求。

`WebAsyncTask` 基本上与 `Callable<T>` 是一样的，它包装了一个 `Callable<T>` 对象，但也提供了一些字段，可以用来指定运行 `Callable<T>` 的 `TaskExecutor`，以及设置一个超时时间，当超过该时间后会自动提交响应。

`DeferredResult` 不会触发执行，相反，当调用 `DeferredResult#setResult` 方法时，Spring MVC 会将 servlet 容器线程返回到线程池中，并且只触发一个异步响应。`DeferredResult` 实例可能会被缓存，并稍后被另一个执行线程更新。举例说明，为了可以被 `@EventListener` 或者 `@RabbitListener` 标注的方法稍后查找到，某个端点可能会对 `DeferredResult` 的引用，存储在一个通过相关 ID 映射的共享 `ConcurrentHashMap<k,v>` 中。一旦获得引用，那些异步线程就可以调用 `DeferredResult#setResult` 方法，并最终通知容器将响应返回给客户端。

我们的示例程序会在一个单独的线程中处理写入操作，这里假设由于缓存的原因，写入操作比读取操作慢。这个示例通过在 `Callable<T>` 中处理上传、写入操作，将它们转移到一个单独的线程中。从服务的角度来看这样做是有效的，但从客户的角度来看无关紧要。只要 `Callable<T>` 正在执行，客户端就会被一直阻塞，等待回复。

另一种方法是立即返回一个 HTTP 202 Accepted 状态码，并提供一个 Location 头，表示新创建资源的获取位置，或者至少可以查询其创建过程中的状态信息。

Google Protocol Buffers

内容协商允许我们尽最大可能去优化，但是另一方面也需要提供更完整的支持。Google Protocol Buffers 就是一个常见的例子。Google Protocol Buffers 是一个高效的、跨平台的通信协议。如果请求的客户端不支持 Google Protocol Buffers，那么根据内容协商，Spring 可以降级为使用 JSON 或 XML。更高效的客户端会得到更快速的响应，但是所有客户端都可以支持，这就是内容协商的原则。

Google Protocol Buffers 是一个高效的序列化格式。Google Protocol Buffers 消息不会携带任何描述消息结构的信息，只包含消息中的数据。消息的结构是通过 Google Protocol



Buffers 的 .proto 模式来定义的。该模式可用来生成与指定语言交互的绑定代码，因此 Google Protocol Buffers 可以为常见的大多数语言和平台提供强大的支持。服务不需要去检查并弄懂每个请求的消息结构，这些在提供模式的时候都已经完成了（如示例 6-3 所示）。

示例6-3 一个Customer消息的Google Protocol Buffers示例

```
package demo;

option java_package = "demo"; ❶

option java_outer_classname = "CustomerProtos"; ❷

message Customer { ❸
    optional int64 id = 1;
    required string firstName = 2;
    required string lastName = 3;
}

message Customers { ❹
    repeated Customer customer = 1;
}
```

- ❶ 针对某种特定语言的包或者模块。
- ❷ 如果你指定了 `java_outer_classname`，那么 Google Protocol Buffers 可以在内部类中嵌套其他类型，因此产生的结果是 `demo.CustomerProtos.Customer`。
- ❸ 单个 Customer 消息的模式。
- ❹ Customer 消息集合的模式。

定义中的字段被赋予了一个整数偏移量，以便帮助编译器读取二进制数据流。如果客户端知道是一个版本较早的消息定义，而服务端回复的是一个最新的消息，那么只要偏移量不变，客户端与服务端之间仍然能够相互通信。这很适合用在分布式系统，因为在分布式系统中，客户端和服务端不可能总是使用相同版本的消息。独立的可部署性，就是在不影响其他服务的情况下单独部署一个服务的能力，这是微服务的一个非常重要的特性。它可以让团队不受其他事情打扰，持续迭代和开发自己的服务。Google Protocol Buffers 支持独立部署这种松耦合的方式。

protoc 编译器用来生成特定于语言（Java、Python、C、.NET、PHP、Ruby 等）的绑定，从而读写 .proto 模式所描述的消息。下面我们为 Customer Google Protocol Buffers 定义编写一段生成 Java、Python 和 Ruby 绑定的脚本。我们可以使用这些绑定与使用 Google



Protocol Buffers 的 REST 端点进行通信（如示例 6-4 所示）。

示例6-4 生成Java、Ruby和Python等客户端脚本

```
#!/usr/bin/env bash

SRC_DIR=`pwd`
DST_DIR=`pwd`/../../src/main/

echo source:          ${SRC_DIR}
echo destination root: ${DST_DIR}

function ensure_implementations(){
    gem list | grep ruby-protocol-buffers || sudo gem install ruby-protocol-buffers
    go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
}

function gen(){
    D=$1
    echo $D
    OUT=$DST_DIR/$D
    mkdir -p $OUT
    protoc -I=$SRC_DIR --${D}_out=$OUT $SRC_DIR/customer.proto
}

ensure_implementations

gen java
gen python
gen ruby
```

通过脚本生成的 Java、Ruby 和 Python 客户端并不是非常复杂，但是它们可以正常工作。示例 6-5 显示了正在运行的 Python 绑定代码。

示例6-5 Python语言的Google Protocol Buffers客户端; customer_pb2是由protoc生成的客户端

```
#!/usr/bin/env python

import urllib

import customer_pb2

if __name__ == '__main__':
    customer = customer_pb2.Customer()
    customers_read = urllib.urlopen('http://localhost:8080/customers/1').read()
    customer.ParseFromString(customers_read)
    print customer
```

示例 6-6 展示了正在运行的 Ruby 绑定。



示例6-6 Ruby语言的Google Protocol Buffers客户端; /customers.pb是由protoc生成的客户端

```
#!/usr/bin/ruby

require './customer.pb'
require 'net/http'
require 'uri'

uri = URI.parse('http://localhost:8080/customers/3')
body = Net::HTTP.get(uri)
puts Demo::Customer.parse(body)
```

Spring MVC 提供了一个自定义的 `HttpMessageConverter` 实现，它可以读写由 `.proto` 定义生成的 Java 绑定对象。`protoc` 编译器会为 Google Protobuf 编译器提供可处理 REST 请求的绑定代码。我们来举一个例子，一个能够处理 `application/x-protobuf` 的 REST API。这个服务的大部分代码我们应该很熟悉。实际上，除了 URI 路径之外，这个示例和之前的主要区别在于，必须从一个 `Customer` 对象转换到 Protobuffer 特定的 `CustomerProtos.Customer DTO` 对象，然后再转换回来（如示例 6-7 所示）。

示例6-7 一个利用内容协商的REST服务，提供媒体类型为`application / x-protobuf`的内容

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

//@formatter:off
import org.springframework.web.servlet.mvc.method
    .annotation.MvcUriComponentsBuilder;
import static org.springframework.web.servlet.support
    .ServletUriComponentsBuilder.fromCurrentRequest;
//@formatter:on

import java.net.URI;
import java.util.Collection;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping(value = "/v1/protos/customers")
public class CustomerProtobufRestController {

    private final CustomerRepository customerRepository;

    @Autowired
    public CustomerProtobufRestController(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }
}
```




```

@GetMapping(value =("/{id}")
ResponseBody<CustomerProtos.Customer> get(@PathVariable Long id) {
    return this.customerRepository.findById(id).map(this::fromEntityToProtobuf)
        .map(ResponseEntity::ok)
        .orElseThrow(() -> new CustomerNotFoundException(id));
}

@GetMapping
ResponseBody<CustomerProtos.Customers> getCollection() {
    List<Customer> all = this.customerRepository.findAll();
    CustomerProtos.Customers customers = this.fromCollectionToProtobuf(all);
    return ResponseEntity.ok(customers);
}

@PostMapping
ResponseBody<CustomerProtos.Customer> post(
    @RequestBody CustomerProtos.Customer c) {

    Customer customer = this.customerRepository.save(new Customer(c
        .getFirstName(), c.getLastName()));

    URI uri = MvcUriComponentsBuilder.fromController(getClass()).path("/{id}")
        .buildAndExpand(customer.getId()).toUri();
    return ResponseEntity.created(uri).body(this.fromEntityToProtobuf(customer));
}

@PutMapping("/{id}")
ResponseBody<CustomerProtos.Customer> put(@PathVariable Long id,
    @RequestBody CustomerProtos.Customer c) {

    return this.customerRepository
        .findById(id)
        .map(
            existing -> {

                Customer customer = this.customerRepository.save(new Customer(existing
                    .getId(), c.getFirstName(), c.getLastName()));

                URI selfLink = URI.create(fromCurrentRequest().toUriString());

                return ResponseEntity.created(selfLink).body(
                    fromEntityToProtobuf(customer));
            }).orElseThrow(() -> new CustomerNotFoundException(id));
}

private CustomerProtos.Customers fromCollectionToProtobuf(
    Collection<Customer> c) {
    return CustomerProtos.Customers
        .newBuilder()
        .addAllCustomer(
            c.stream().map(this::fromEntityToProtobuf).collect(Collectors.toList()))

```



```

    .build();
}

private CustomerProtos.Customer fromEntityToProtobuf(Customer c) {
    return fromEntityToProtobuf(c.getId(), c.getFirstName(), c.getLastName());
}

private CustomerProtos.Customer fromEntityToProtobuf(Long id, String f,
String l) {
    CustomerProtos.Customer.Builder builder = CustomerProtos.Customer
        .newBuilder();
    if (id != null && id > 0) {
        builder.setId(id);
    }
    return builder.setFirstName(f).setLastName(l).build();
}
}

```

我们需要注册一个自定义的 `HttpMessageConverter` 实现，来通知 Spring MVC 使用新的内容类型，如示例 6-8 所示。

示例6-8 注册一个自定义的`HttpMessageConverter`实现，其能够将HTTP消息与`application/x-protobuf`类型进行转换

```

package demo;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.converter.protobuf.ProtobufHttpMessageConverter;

@Configuration
class GoogleProtocolBuffersConfiguration {

    @Bean
    ProtobufHttpMessageConverter protobufHttpMessageConverter() {
        return new ProtobufHttpMessageConverter();
    }
}

```

`RestTemplate` 客户端支持与 Spring MVC 服务一样的内容协商机制，因此如果要读取 `application/x-protobuf` 类型的数据，需在 `RestTemplate` 上配置一个 `ProtobufHttpMessageConverter`。

错误处理

`CustomerRestController` 中各种处理器方法可以对现有的记录进行操作，如果没有找到该记录，则抛出异常。这些处理器方法可以返回一个状态码为 404 的 `ResponseEntity` 对象，但是这种错误处理逻辑很快会在多个处理器方法中重复出现。因此，我们需要能够



集中式地处理错误。规模来自于一致性，而一致性来自于自动化。Spring MVC 支持在处理器方法上使用 `@ExceptionHandler` 注解，来监听和响应 Spring MVC 控制器中的错误条件。通常情况下，`@ExceptionHandler` 标记的处理器也位于同一控制器中，因为它们可能会抛出异常。但是，这些异常处理器不能在多个控制器之间共享。如果要集中处理异常逻辑，使用 `@ControllerAdvice` 组件。`@ControllerAdvice` 是一种特殊类型的组件，可以为任意数量的控制器注入行为（并响应异常）。它们是集中处理 `@ExceptionHandler` 的天然场所。

错误是一个有效 API 的重要组成部分。错误应该唯一、简洁地向自动化客户端指出错误情况，并为最终解决问题的人提供支持，至少要让他们了解错误的含义。因此，错误应该尽可能地提供帮助。HTTP 状态代码中包含了更多内容，但并不是特别有用。常见的做法是发送一个错误代码和错误名称，以及一个可读的错误详细内容。现在除了 HTTP 状态代码之外，还没有官方的错误编码方式，但事实上的标准是 `application/vnd.error` 内容类型（<https://github.com/blongden/vnd.error>）。Spring HATEOAS 支持这种错误形式。我们只需将 `org.springframework.boot:spring-boot-starter-hateoas` 添加到类路径中。Spring HATEOAS 提供了单个错误的封装对象 `VndError` 和一组错误的封装对象 `VndErrors`。

我们来看示例 6-9，它演示了一个简单的 `@ControllerAdvice` 方法，拦截所有抛出的异常，并正确返回一个 HTTP 状态码和一个 `VndError` 对象。

示例6-9 使用 `@ControllerAdvice` 进行集中式的错误处理

```
package demo;
```

```
import org.springframework.hateoas.VndErrors;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestController;
```

```
import java.util.Optional;
```

```
@ControllerAdvice(annotations = RestController.class)
public class CustomerControllerAdvice {
```

❶

```
    private final MediaType vndErrorMediaType = MediaType
        .parseMediaType("application/vnd.error");
```

❷

```
    @ExceptionHandler(CustomerNotFoundException.class)
    ResponseEntity<VndErrors> notFoundException(CustomerNotFoundException e) {
```



```

    return this.error(e, HttpStatus.NOT_FOUND, e.getCustomerId() + "");
}

@ExceptionHandler(IllegalArgumentException.class)
ResponseBody<VndErrors> assertionException(IllegalArgumentException ex) {
    return this.error(ex, HttpStatus.NOT_FOUND, ex.getLocalizedMessage());
}

③
private <E extends Exception> ResponseEntity<VndErrors> error(E error,
    HttpStatus httpStatus, String logref) {
    String msg = Optional.of(error.getMessage()).orElse(
        error.getClass().getSimpleName());
    HttpHeaders httpHeaders = new HttpHeaders();
    httpHeaders.setContentType(this.vndErrorMediaType);
    return new ResponseEntity<>(new VndErrors(logref, msg), httpHeaders,
        httpStatus);
}
}

```

- ❶ application/vnd.error 没有内置的媒体类型，所以我们需要自己创建一个。
- ❷ 在这个类中有两个处理器，它们都会响应其他组件中可能抛出的异常。这个处理器会专门用来响应 CustomerNotFoundException（该异常继承自 RuntimeException）。
- ❸ error 方法会创建一个包含 VndErrors 数据的 ResponseEntity 对象，并返回所需的媒体类型和状态码。简单并且有效。

超媒体

如果客户端知道要调用哪个端点，何时去调用它，以及使用何种 HTTP 方法进行调用，那么就说明我们构建了一个不错的 API。HTTP OPTIONS 方法帮助我们弥补了最后一点：让我们知道可以调用指定资源的哪个 HTTP 方法。但是，它并不能提供其他的内容，而我们又希望能看到相关的具体文档。文档的问题在于很少有人能持续更新它，而能阅读文档的人就更少了。它最终无法与代码中所定义的服务规范同步。

Fielding 博士的论文主要是宣传超媒体。超媒体指的是，日常资源中的链接（HTML 标记中的元素）通过向客户端（HTML 浏览器及其用户）提供信息，最终导致应用程序状态的变化。你可以在亚马逊网站上进行这样的尝试，首先输入搜索内容，找到满意的产品进行购买，然后单击并将它添加到购物车中，选择结算并最终付款，所有这些流程都是从一个 HTTP 资源导航到另一个 HTTP 资源。最终，你已经改变了系统的状态。这些链接会告诉你导航到哪里，并且只有在需要的时候才会出现（对于还没有付款的产品则不会有退款链接！），所以这意味着导航的时机。这些从一个资源到另一个资源的步骤代表了一个协议：在软件中，众多步骤和交互都是为了到达一个终点。这个系统之所以

有效，是因为我们人类可以解析链接，提取出我们需要的信息，并且根据网站设计的视觉线索，来理解一系列步骤之间的相关性。

另一方面，机器客户端不像人类那样聪明。人类可以单击由元素 `<a/>` 可视化呈现的链接，而自动化的客户端只能遍历 `<link/>` 元素（虽然人类无法单击它们）。这些链接有两个重要的属性：`rel` 和 `href`。该元素有很多用处，但是最常见的是用来加载样式表数据，如示例 6-10 所示。

示例6-10 加载文档的样式表

```
<link rel="stylesheet" href="bootstrap.css">
```

这些链接提供了所指向资源的元数据。浏览器会首先查看 `rel` 属性，判断链接的资源类型。它可以选择在 `href` 之后加载所链接的资源。客户端使用 `rel` 属性来确定链接资源的相关性。链接资源可以存放在其他任何地方！浏览器不会对资源的位置作出任何假设，它只会使用 `href` 属性所指定的位置。因此，客户端与资源的位置实现了解耦。

即使 API 中的资源位置发生了变化，客户端（浏览器）仍然可以使用 API。客户端可以通过检查 `rel` 属性获得资源的相关性，从而决定去链接哪个资源。这样，`rel` 就变成了一种契约：只要它保持稳定，客户端就永远不会失去链接。

我们来回顾一下通过 Amazon.com API 结算购物的过程。虽然，例子中的 `<link/>` 数据是一个 XML 元素，但通过链接的形式，也可以传输其他编码的数据（例如 JSON）。在 JSON 中，甚至有一个描述这些链接元素的事实上的标准编码。HAL 或超文本应用程序语言（http://stateless.co/hal_specification.html）是 JSON 的一个规范，其内容类型为 `application/hal+json`。所以，你不需要花费时间建立专门的结构来描述 REST 资源，可以直接选择使用 HAL 来实现 HATEOAS 风格（如示例 6-11 所示）。



关于超媒体其实还有其他标准，只是 HAL 机缘巧合受到了欢迎，并且它的确很实用。

示例6-11 第一次使用HAL来描述Customer REST资源

```
{
  "id":1,
  "firstName":"Mark",
  "lastName":"Fisher",
  "_links": {
    "self": {"href":"http://localhost:8080/v2/customers/1"},
    "profile-photo": {"href":"http://localhost:8080/customers/1/photo/"}
  }
}
```

任何兼容 HAL 的客户端都可以使用这样的 API。Spring Boot 支持一个非常方便的 HAL 客户端，我们称之为 HAL 浏览器。可以通过添加 `org.springframework.boot:spring-boot-starter-actuator` 和 `org.springframework.data:spring-data-rest-hal-browser` 依赖项，将它添加到任何 Spring Boot Web 应用程序中。它会注册一个 Spring Boot Actuator 端点，所以你也需要依赖 Actuator（如图 6-1 所示）。

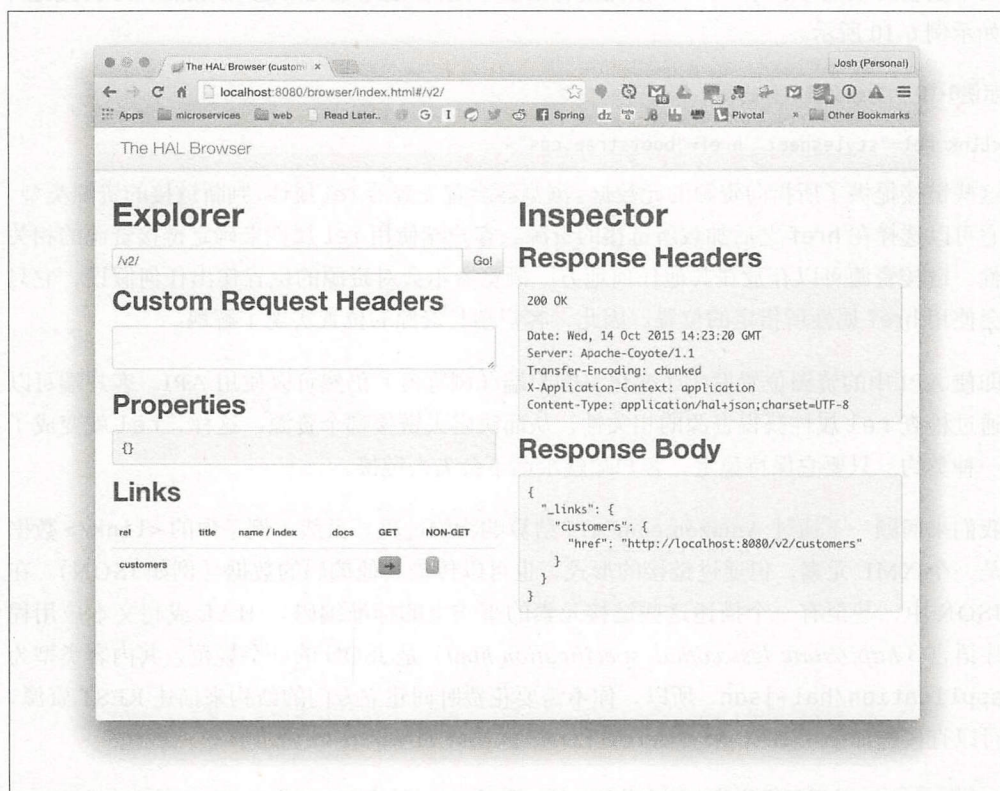


图6-1 采用默认配置的HAL浏览器的Actuator端点

Spring HATEOAS (<http://spring.io/projects/spring-hateoas>) 位于 Spring MVC 的顶层，其提供了必备工具来使用和描述资源中的数据和相关链接。它依赖于 Spring MVC。事实上，我们不是消费或生产某个 T 类型的资源，而是对于单个对象来说，我们消费或生产一个 `org.springframework.hateoas.Resource<T>` 类型的资源，对于对象集合来说，则是一个 `org.springframework.hateoas.Resources<T>` 类型的资源。

在 Spring HATEOAS 中，Resource 是一个包含数据和一组相关链接的封装对象。你会经常需要将 T 类型的对象转换为 `Resource<T>` 或 `Resources<T>` 类型的对象。通过 Spring HATEOAS 的 `org.springframework.hateoas.ResourceAssembler` 实例，我们可以简

化这个过程。让我们重新回到 CustomerRestController 类，为它添加超媒体和 Spring HATEOAS（如示例 6-12 所示）。

示例6-12 修改后的CustomerHypermediaRestController

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

//@formatter:off
import org.springframework.web.servlet.mvc.method
    .annotation.MvcUriComponentsBuilder;
import org.springframework.web.servlet.support
    .ServletUriComponentsBuilder;
//@formatter:on

import java.net.URI;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

❶
@RestController
@RequestMapping(value = "/v2", produces = "application/hal+json")
public class CustomerHypermediaRestController {

    private final CustomerResourceAssembler customerResourceAssembler; ❷

    private final CustomerRepository customerRepository;

    @Autowired
    CustomerHypermediaRestController(CustomerResourceAssembler cra,
                                      CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
        this.customerResourceAssembler = cra;
    }

    ❸
    @GetMapping
    ResponseEntity<Resources<Object>> root() {
        Resources<Object> objects = new Resources<>(Collections.emptyList());
        URI uri = MvcUriComponentsBuilder
            .fromMethodCall(MvcUriComponentsBuilder.on(getClass()).getCollection())
            .build().toUri();
        Link link = new Link(uri.toString(), "customers");
        objects.add(link);
    }
}
```

```

return ResponseEntity.ok(objects);
}

④
@GetMapping("/customers")
ResponseEntity<Resources<Resource<Customer>>> getCollection() {
    List<Resource<Customer>> collect = this.customerRepository.findAll().stream()
        .map(customerResourceAssembler::toResource)
        .collect(Collectors.<Resource<Customer>>toList());
    Resources<Resource<Customer>> resources = new Resources<>(collect);
    URI self = ServletUriComponentsBuilder.fromCurrentRequest().build().toUri();
    resources.add(new Link(self.toString(), "self"));
    return ResponseEntity.ok(resources);
}

@RequestMapping(value = "/customers", method = RequestMethod.OPTIONS)
ResponseEntity<?> options() {
    return ResponseEntity
        .ok()
        .allow(HttpMethod.GET, HttpMethod.POST, HttpMethod.HEAD, HttpMethod.OPTIONS,
            HttpMethod.PUT, HttpMethod.DELETE).build();
}

@GetMapping(value = "/customers/{id}")
ResponseEntity<Resource<Customer>> get(@PathVariable Long id) {
    return this.customerRepository.findById(id)
        .map(c -> ResponseEntity.ok(this.customerResourceAssembler.toResource(c)))
        .orElseThrow(() -> new CustomerNotFoundException(id));
}

@PostMapping(value = "/customers")
ResponseEntity<Resource<Customer>> post(@RequestBody Customer c) {
    Customer customer = this.customerRepository.save(new Customer(c
        .getFirstName(), c.getLastName()));
    URI uri = MvcUriComponentsBuilder.fromController(getClass())
        .path("/customers/{id}").buildAndExpand(customer.getId()).toUri();
    return ResponseEntity.created(uri).body(
        this.customerResourceAssembler.toResource(customer));
}

@DeleteMapping(value = "/customers/{id}")
ResponseEntity<?> delete(@PathVariable Long id) {
    return this.customerRepository.findById(id).map(c -> {
        customerRepository.delete(c);
        return ResponseEntity.noContent().build();
    }).orElseThrow(() -> new CustomerNotFoundException(id));
}

@RequestMapping(value = "/customers/{id}", method = RequestMethod.HEAD)
ResponseEntity<?> head(@PathVariable Long id) {
    return this.customerRepository.findById(id)
        .map(exists -> ResponseEntity.noContent().build())
        .orElseThrow(() -> new CustomerNotFoundException(id));
}

```



```

}

@PutMapping("/customers/{id}")
ResponseBody<Resource<Customer>> put(@PathVariable Long id,
    @RequestBody Customer c) {
    Customer customer = this.customerRepository.save(new Customer(id, c
        .getFirstName(), c.getLastName()));
    Resource<Customer> customerResource = this.customerResourceAssembler
        .toResource(customer);
    URI selfLink = URI.create(ServletUriComponentsBuilder.fromCurrentRequest()
        .toUriString());
    return ResponseEntity.created(selfLink).body(customerResource);
}
}

```

- ❶ 我们发送的不是一个 `application/json` 类型的响应，而是一个 `application/hal+json` 媒体类型的响应。
- ❷ 注入了一个自定义的 `ResourceAssembler` 实现，来简化从类型 `T` 到 `Resource<T>` 的转换过程。
- ❸ 根端点只是返回了一组 `Link` 元素，作为一种导航菜单使用：它应该可以无师自通地从 / 开始导航。
- ❹ `Customers` 集合上的 `GET` 方法和以前的流程一样，但是我们在资源中添加了一个 `Spring HATEOAS Link`。Link 是通过 `Spring MVC` 的工具类 `ServletUriComponentsBuilder` 来构造的。

该类的其余部分基本上和以前的一样。`CustomerHypermediaRestController` 中的大部分方法依靠注入的 `CustomerResourceAssembler` 来完成。`CustomerResourceAssembler` 的定义如示例 6-13 所示。

示例6-13 `CustomerResourceAssembler`为指定的Customer实体提供了一组默认的连接

```

package demo;

import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resource;
import org.springframework.hateoas.ResourceAssembler;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.mvc.method.annotation
    .MvcUriComponentsBuilder;

import java.net.URI;

@Component
class CustomerResourceAssembler implements

```

```

ResourceAssembler<Customer, Resource<Customer>> {

@Override
public Resource<Customer> toResource(Customer customer) {

    Resource<Customer> customerResource = new Resource<>(customer);
    URI photoUri = MvcUriComponentsBuilder
        .fromMethodCall(
            MvcUriComponentsBuilder.on(CustomerProfilePhotoRestController.class).read(
                customer.getId()))
        .buildAndExpand().toUri();

    URI selfUri = MvcUriComponentsBuilder
        .fromMethodCall(
            MvcUriComponentsBuilder.on(CustomerHypermediaRestController.class).get(
                customer.getId()))
        .buildAndExpand().toUri();

    customerResource.add(new Link(selfUri.toString(), "self"));
    customerResource.add(new Link(photoUri.toString(), "profile-photo"));
    return customerResource;
}
}

```

ResourceAssembler 能够根据实体的状态，选择性地包含或排除一些链接。在这里，我们会检查是否存在个人资料照片，如果有则选择性地包含照片的链接。这些可用的链接描述了客户端与系统交互的协议。

结果中包含了超媒体以及 HTTP（和 HTTP OPTIONS 方法）提供的线索，这使得我们可以在没有任何文档的情况下，依然轻松地浏览这个 API。

HAL 超媒体只是众多流行方案之一。2009 年推出的 Web 应用描述语言 (<https://www.w3.org/Submission/wadl/>)，也是一种广泛用于描述 HTTP 服务的方式。ALPS (<http://alps.io/spec/>) 是最近才推出的方案，在媒体类型不确定的情况下，它可以同时定义可能的状态转换，以及状态转换过程中所涉及的资源属性。Spring Data REST 为 ALPS 提供了自动支持。

媒体类型和模式

我们已经了解了如何使用 Spring HATEOAS 在 REST API 中引入超媒体。我们的目标是向客户端提供尽可能多的信息，以便它能够直接自行浏览 API。HTTP 已经规范了可能出现的动词（PUT、POST、GET、DELETE 等），而超媒体可以告诉我们去哪里找到指定的资源。

HTTP 提供了内容类型的概念。从理论上讲，内容类型为任何客户端都提供了足够的信息，由它足以了解服务的内容是什么，以及如何操纵它。但是在实践中，这是不够的。内容类型 image/jpg 告诉我们，我们可以使用 JPEG 图像阅读器解码服务端返回的字节。内

容类型 `application/json` 或 `application/hal+json` 告诉我们，可以使用 JSON 阅读器读取从资源返回的字节，并且其中可能还包含 HAL 编码的链接。但是，这些内容类型不会告诉我们，某个 JSON 文档是否与指定的结构相匹配。为此，我们需要使用模式的概念。

XML 提供了 XML 模式 (<http://www.w3.org/XML/Schema>) 来限制 XML 资源的结构。Google Protocol Buffer 也有一个模式的定义。对于 JSON 来说，虽然没有那么简单，但是有许多事实上的选择，例如 JSON 模式 (<http://json-schema.org>)。模式提供了一种定义数据结构的方式。

API 版本

我们唯一确定的就是事情会改变，这是不可避免的。事实上，云原生架构最大的好处之一就是它鼓励小批量更改，就如同微服务一样，可以快速并独立地发展。变化是一个好事，它意味着进化！我们希望拥抱变化，但是我们不希望改变 API，或者损害到系统或第三方客户端的利益。我们必须找到相应的方法来发展服务，但是又不会影响到客户端。

我们可以设法不造成破坏。这听上去有点像通过拒绝发布来降低发布的风险一样可笑。但是我们的确可以做一些努力。如果你只是更改了实现的细节，那么不应该对 API 的接口形式造成改变。我们应该尽早发现任何可能破坏接口的潜在风险。消费者驱动的契约测试有助于保护 API 免受破坏（有关这方面的更多信息，请参考第 4 章中的讨论）。持续集成也提供了一个有用的反馈循环，其可以使用尽可能多的客户端对更改后的 API 进行测试，来不断避免 API 接口遭到意外破坏。

不过，事情总会发生变化。客户端和服务应该提供一定的灵活性，来降低对变化的敏感度。Martin Fowler 在他的博客中谈到了宽容阅读器的想法 (<https://martinfowler.com/bliki/TolerantReader.html>)，即一个避免对数据更改过度敏感的 API 客户端。假如客户端希望查找 `order-id` 这个 XML 元素或 JSON 元素，那么可以通过 JSON-Path 或者 XPath 查询，让客户端按照原结构的顺序和层次找到匹配元素。

Postel 法则也被称为鲁棒性原则 (https://en.wikipedia.org/wiki/Robustness_principle)，他们认为服务应该对自己的输出严格，对别人的输入宽容。如果一个服务只需要使用消息的一部分数据，那么它就不应该理会是否可能含有其他信息。

只要变化是渐进式和附加式的，而不是破坏式的，那么服务就容易变得更加健壮。如果服务版本 1 中出现的元素在版本 2 中突然消失，则客户端几乎肯定会受到影响。一个服务应该清楚哪些客户端能够工作。一种方法是使用语义版本控制。语义版本是 MAJOR, MINOR, PATCH 的其中一种形式。MAJOR 版本号只有在 API 不兼容以前的版本时才会更改。

MINOR 版本号应该在 API 发生改变但是现有客户端还能够继续使用的环境下更改。PATCH 版本号的更改表示对现有功能错误的修复。

语义版本控制会告诉客户端可以使用什么版本的 API，但即使客户端知道该版本，它们也可能没有做好使用新版本的准备。如果客户端在服务升级后需要被迫升级，那么这就威胁到了微服务的根基之一——独立发展的能力。因此，有的时候你需要同时管理多个 API 版本。一种方法是部署和维护两个不同的代码库，但这可能很快就会给维护带来一场噩梦。相反，你应该考虑在同一个代码库中提供不同版本的 API。如果可能的话，可以尝试将发往旧端点的请求，转移到新的端点，这样所有请求最终都在同一个地方进行处理，从而减轻测试的负担。

客户端需要告诉服务端，它要与哪个版本的 API 进行通信。在 REST API 中有一些常见的方法：在 URL 中对版本号进行编码，或者将版本号编码到任意（并且私有）的 HTTP 头信息中，或者将版本号编码到请求 Accept 头信息的内容类型中（如示例 6-14 所示）。

示例6-14 VersionedRestController演示了几种定义REST API版本的方法

```
package demo;

import org.springframework.web.bind.annotation.*;

//@formatter:off
import static org.springframework.http
    .MediaType.APPLICATION_JSON_VALUE;
//@formatter:on

@RestController
@RequestMapping("/api")
public class VersionedRestController {

    //@formatter:off
    public static final String V1_MEDIA_TYPE_VALUE
        = "application/vnd.bootiful.demo-v1+json";

    public static final String V2_MEDIA_TYPE_VALUE
        = "application/vnd.bootiful.demo-v2+json";
    //@formatter:on

    private enum ApiVersion {
        v1, v2
    }

    public static class Greeting {

        private String how;

        private String version;
```



```

public Greeting(String how, ApiVersion version) {
    this.how = how;
    this.version = version.toString();
}

public String getHow() {
    return how;
}

public String getVersion() {
    return version;
}

①
@GetMapping(value = "{version}/hi", produces = APPLICATION_JSON_VALUE)
Greeting greetWithPathVariable(@PathVariable ApiVersion version) {
    return greet(version, "path-variable");
}

②
@GetMapping(value = "/hi", produces = APPLICATION_JSON_VALUE)
Greeting greetWithHeader(@RequestHeader("X-API-Version") ApiVersion version) {
    return this.greet(version, "header");
}

③
@GetMapping(value = "/hi", produces = V1_MEDIA_TYPE_VALUE)
Greeting greetWithContentNegotiationV1() {
    return this.greet(ApiVersion.v1, "content-negotiation");
}

④
@GetMapping(value = "/hi", produces = V2_MEDIA_TYPE_VALUE)
Greeting greetWithContentNegotiationV2() {
    return this.greet(ApiVersion.v2, "content-negotiation");
}

private Greeting greet(ApiVersion version, String how) {
    return new Greeting(how, version);
}
}

```

- ① 该方法在 URL 中对版本号进行编码；Spring MVC 会自动将 URL 参数映射到对应的 `ApiVersion` 枚举实例。可以使用 curl `http://localhost:8080/api/V2/hi` 进行测试。
- ② 该方法会自动将请求头信息转换为相应的 `ApiVersion` 枚举实例。可以使用 curl `-H "X-API-Version: V1" http://localhost:8080/api/hi` 进行测试。
- ③ 该控制器中的处理器方法会处理媒体类型为 `application/vnd.bootiful.demo-v1`

+json 的请求。可以使用 `curl -H "Accept: application/vnd.bootiful.demo-v1+json" http://localhost:8080/api/hi` 进行测试。

- ④ 该控制器中的处理器方法会处理媒体类型为 `application/vnd.bootiful.demo-v2+json` 的请求。可以使用 `curl -H "Accept: application/vnd.bootiful.demo-v2+json" http://localhost:8080/api/hi` 进行测试。

编写 REST API 文档

敏捷宣言鼓励可运行的软件，而不是全面的文档。这也就是说，先让软件能够工作，再用文档把它记录下来。一个能够正确解释自己、可运行的软件，要比内容过时的文档好得多。代码是一种活生生的、会呼吸的东西，正如我们认为大多数人在某些时候都经历的，与代码开发同时编写的文档往往逐渐会与实际的代码不同步。文档只是一个地图，但代码才是实际的领土。如果你发现自己迷路了，应更熟悉你的领土而不是地图！

当然，文档仍然是一个非常自然的、方便的东西。好的是，我们的工程师已经做了大量的工作，来将 API 文档与 API 本身结合起来。JavaDoc 是第一个被大规模使用的工具，尽管其他语言早已引入了其他方案。因为 JavaDoc 与代码在一起，所以由开发人员负责维护。我们希望的是，因为 JavaDoc 与代码在一起，所以开发人员总是能够及时维护文档，从而反映代码的真实性。当然，我们知道这其实很难做到！除了 JavaDoc 以外，还有一些复杂的工具可以根据代码来生成文档。

超媒体为设计良好的 API 提供了许多线索。HTTP `OPTION` 请求可以告诉我们，一个指定的资源能支持哪些 HTTP 动词。超媒体链接可以告诉我们，一个资源有什么样的关联关系。模式可以告诉我们某个资源的数据结构形式。但是，我们没有很好的方式能够说明 HTTP 请求参数或者头信息，也没有很好的方式来解释 API 的动机或意图。最好的文档应该是说明目的，而不是说明如何实现。我们仍然缺少一种方式，能够尽可能自动和全面地完成 API 文档，并且保证文档与代码始终一致。

现在已经出现了一些解决类似问题的工具，例如 Swagger。Swagger 需要侵入代码本身，它依靠在 API 注解中嵌入字符串来生成文档。但是，手工维护文档内容依然很麻烦，而且有时无法充分理解 API 的意图，因为它试图自动将构造函数映射到 HTTP 协议上。

Spring REST Docs (<http://spring.io/projects/spring-restdocs>) 采用了不同的方式。它作为 Spring MVC 测试框架的一个外部框架，允许使用 Asciidoctor 标记语言，像编写一本书一样地去编写文档！

我们需要将 Spring REST Docs 作为 test 作用域的依赖项添加到类路径中：`org`。

springframework.restdocs:spring-restdocs-mockmvc。然后，可以通过测试来为 API 生成文档。我们试图捕获测试中确定的交互结果。我们来看一下示例 6-15。

示例6-15 ApiDocumentation.java使用 Spring MVC 测试 API

```
package demo;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;

import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import javax.servlet.RequestDispatcher;

import static org.hamcrest.Matchers.is;
import static org.hamcrest.Matchers.notNullValue;
import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.
    get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.
    print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
    jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
    status;
//@formatter:off
❶
@AutoConfigureRestDocs(outputDir = "target/generated-snippets")
@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class,
    webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
public class ApiDocumentation {
//@formatter:off

❷
// @Rule public final RestDocumentation restDocumentation =
//     new RestDocumentation(
//         "target/generated-snippets");

@Autowired
private MockMvc mockMvc;

@Test
public void errorExample() throws Exception {
    this.mockMvc
```

```

    .perform(
        get("/error")
        .contentType(MediaType.APPLICATION_JSON)
        .requestAttr(RequestDispatcher.ERROR_STATUS_CODE, 400)
        .requestAttr(RequestDispatcher.ERROR_REQUEST_URI, "/customers")
        .requestAttr(RequestDispatcher.ERROR_MESSAGE,
            "The customer 'http://localhost:8443/v1/customers/123' does not exist"))
    .andDo(print()).andExpect(status().isBadRequest())
    .andExpect(jsonPath("error", is("Bad Request")))
    .andExpect(jsonPath("timestamp", is(notNullValue()))))
    .andExpect(jsonPath("status", is(400)))
    .andExpect(jsonPath("path", is(notNullValue()))))
    .andDo(document("error-example")); ❸
}

@Test
public void indexExample() throws Exception {
    this.mockMvc.perform(get("/v1/customers")).andExpect(status().isOk())
        .andDo(document("index-example"));
}
}

```

- ❶ 测试规则支持对测试进行切面处理。在示例中运行测试时，RestDocumentation 规则会在 target/generated-snippets 目录下生成 Asciidoctor 文档片段。
- ❷ 静态方法 documentationConfiguration 只是简单地注册了 RestDocumentation 和 MockMvc 对象，供测试用例来调用 API。
- ❸ 传递给 document 方法的 error-example 字符串，定义了该测试生成自动文档的逻辑标识符。在该示例中，生成的文档会被保存在一个名为 error-example 的文件夹中，其中会存在多个 .adoc 文件，例如 curl-request.adoc、http-request.adoc、http-response.adoc 等。这些文档片段会被用来创建一个更完整的 Asciidoctor 文档。

通常，测试插件会查找以 Test 结尾的类，并运行它们。这里，我们需要配置测试插件包含和运行所有以 *Documentation 结尾的类（如示例 6-16 所示）。

示例6-16 在Maven Surefire插件中包含*Documentation.java

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <includes>
      <include>/**/*.Documentation.java</include>
    </includes>
  </configuration>
</plugin>

```


这些测试运行的同时，将自动生成 AsciiDoctor 文档片段，它们会被包含在一个更大的 AsciiDoctor 文档中（如示例 6-17 所示）。

示例6-17 运行测试后自动生成的AsciiDoctor片段

```
.
├─ customer-get-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   ├── links.adoc
│   └── response-fields.adoc
├─ customer-update-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   └── request-fields.adoc
├─ customers-create-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   ├── links.adoc
│   ├── request-fields.adoc
│   └── response-fields.adoc
├─ customers-list-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   └── response-fields.adoc
├─ error-example
│   ├── curl-request.adoc
│   ├── http-request.adoc
│   ├── http-response.adoc
│   └── response-fields.adoc
└─ index-example
    ├── curl-request.adoc
    ├── http-request.adoc
    └── http-response.adoc
```

6 directories, 26 files

这个方法因工具不同或有差异，Spring RESTDocs 项目提供了更详细的示例，但你可以很容易地使用 Maven 的 `org.asciidoctor:asciidoctor-maven-plugin` 插件来处理 AsciiDoctor 文档，如示例 6-18 所示。

示例6-18 AsciiDoctor Maven插件

```
<plugin>
  <groupId>org.asciidoctor</groupId>
  <artifactId>asciidoctor-maven-plugin</artifactId>
  <version>1.5.2</version>
  <executions>
    <execution>
```

```

<id>generate-docs</id>
<phase>prepare-package</phase>
<goals>
  <goal>process-asciidoc</goal>
</goals>
<configuration>
  ❶
  <backend>html</backend>
  <doctype>book</doctype>
  <attributes>
    <snippets>${project.build.directory}/generated-snippets</snippets> ❷
  </attributes>
</configuration>
</execution>
</executions>
</plugin>

```

- ❶ 根据配置，应用程序将在 `src/main/resources` 目录中查找所有有效的 `.adoc` 文件，然后将它们转换为 `.pdf` 和 `.html` 文件。
- ❷ 为了简化解析代码片段操作，并且避免在文档中出现重复，暴露一个属性，解析到 `generated-snippets` 目录。

这里不会贴出太多 Asciidoctor 示例的内容，你可以参考随书代码，但是你应该清楚，我们可以很容易将这些片段整合到一个 Asciidoctor 文档中。例如，使用如下命令就可以引入 `response-fields.adoc` 片段：`include :: {snip pets} /error-example/response-fields.adoc []`。

一旦你完成了文档编写，并适当使用了生成的代码片段，那么文档就可以作为应用程序本身的一部分服务资源。可以通过配置构建工具，将生成的文件复制到 Spring Boot 的 `src/main/resources/static` 目录。示例 6-19 展示了 Maven 的配置。

示例6-19 配置Maven资源插件，将生成的文档包含在Spring Boot的静态目录中，以便可以通过`http://localhost:8080/docs`访问

```

<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <executions>
    <execution>
      <id>copy-resources</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>copy-resources</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```



```

<configuration>
<outputDirectory>${project.build.outputDirectory}/static/docs</outputDirectory>
<resources>
  <resource>
    <directory>${project.build.directory}/generated-docs</directory>
  </resource>
</resources>
</configuration>
</execution>
</executions>
</plugin>

```

这样的结果，就是能让文档始终保持更新，与代码保持一致，并保证构建成功。

客户端

在本节中，我们将以交互和编程两种方式来了解如何使用 REST API。

用于临时浏览和交互的 REST 客户端

Spring Boot 简化了 HAL 浏览器调用 HAL REST API 的工作。有许多同类、免费支持调用 REST API 的工具。以下是一些在开发过程中使用方便的工具：

Firefox 的 Poster 插件

这个免费的 Firefox 插件是一个很方便的工具，位于浏览器的右下角，显示为一个黄色的小图标。单击它将打开一个对话框，可以在其中描述和执行 HTTP 请求。它可以很方便地处理文件上传和内容协商的事情（如图 6-2 所示）。

curl

古老的 curl 命令是一个命令行工具，它支持任何方式的 HTTP 请求，可以使用它方便地发送 HTTP 头信息、自定义请求内容等（如图 6-3 所示）。



笔者自己也是 HTTPIe 的粉丝（<https://httpie.org>）。

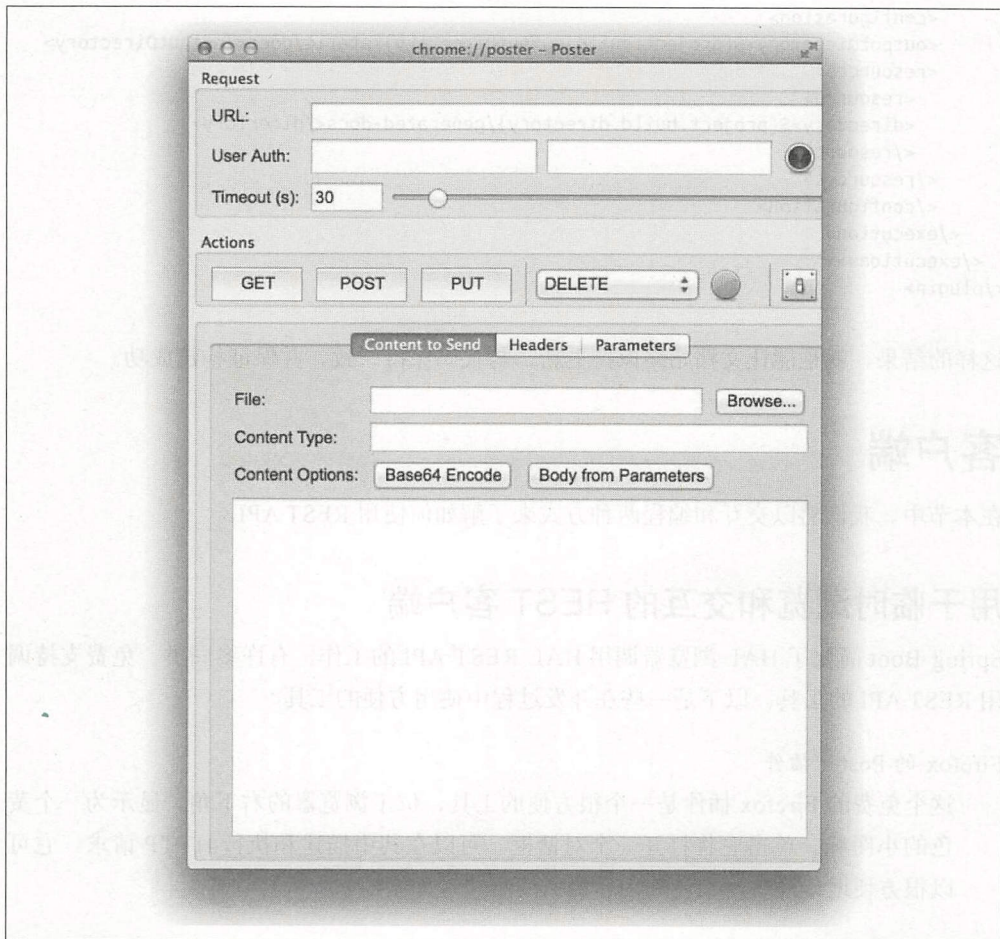


图6-2 Firefox的Poster插件提供了很多方便的功能



图6-3 一个使用curl的脚本

Google Chrome 高级 HTTP 客户端扩展

该扩展允许你描述和执行 HTTP 请求，并将配置保存下来供以后重复使用（如图 6-4 所示）。

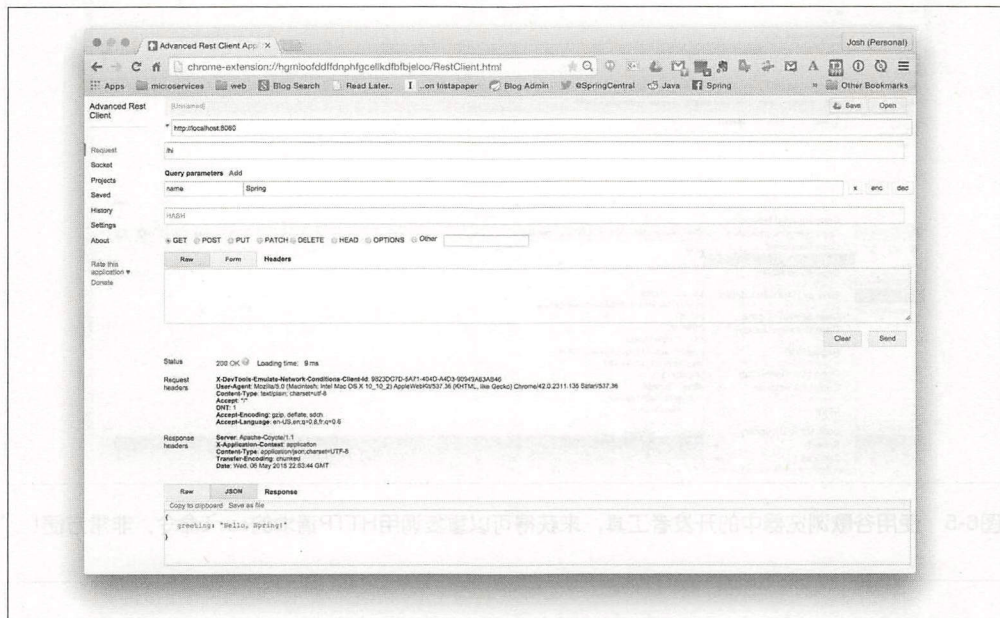


图6-4 Google Chrome高级HTTP客户端

高级 HTTP 客户端

该插件可以与 Google Chrome 开发者工具很好地集成在一起。你可以打开开发者工具（每个 Google Chrome 安装中）并查看 Network 标签页。从那里，很容易找到使用高级 HTTP 客户端编写和触发的请求，然后将其导出为 curl 命令（如图 6-5 所示）。

PostMan

这个方便的 Chrome 扩展，提供了保存 HTTP 查询、跨设备同步以及 HTTP 请求分组等功能。它是在正式工作中使用的 HTTP 客户端之一（如图 6-6 所示）。

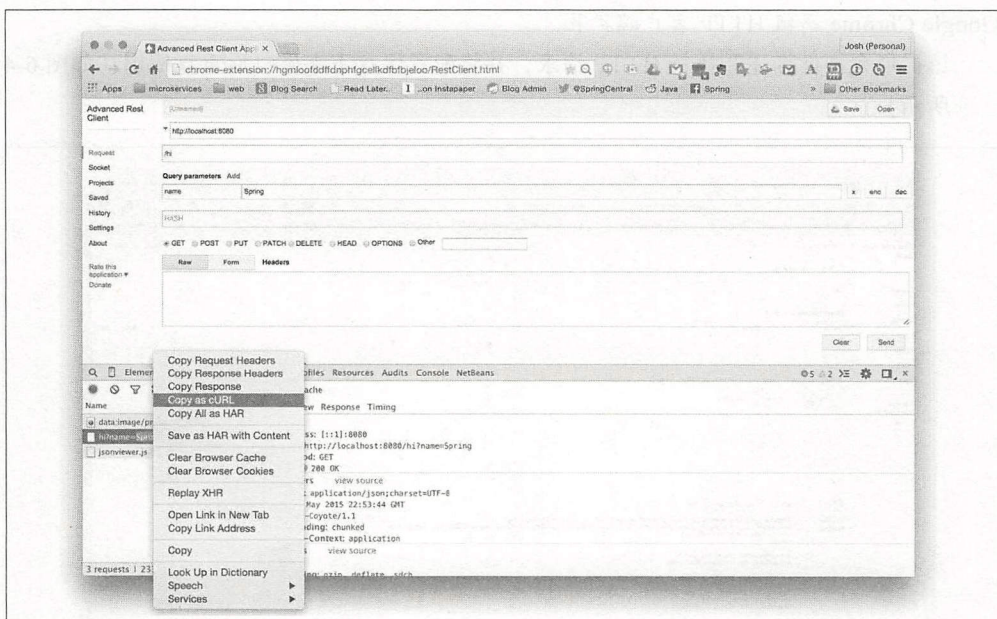


图6-5 使用谷歌浏览器中的开发者工具，来获得可以重复调用HTTP请求的curl命令。非常方便！

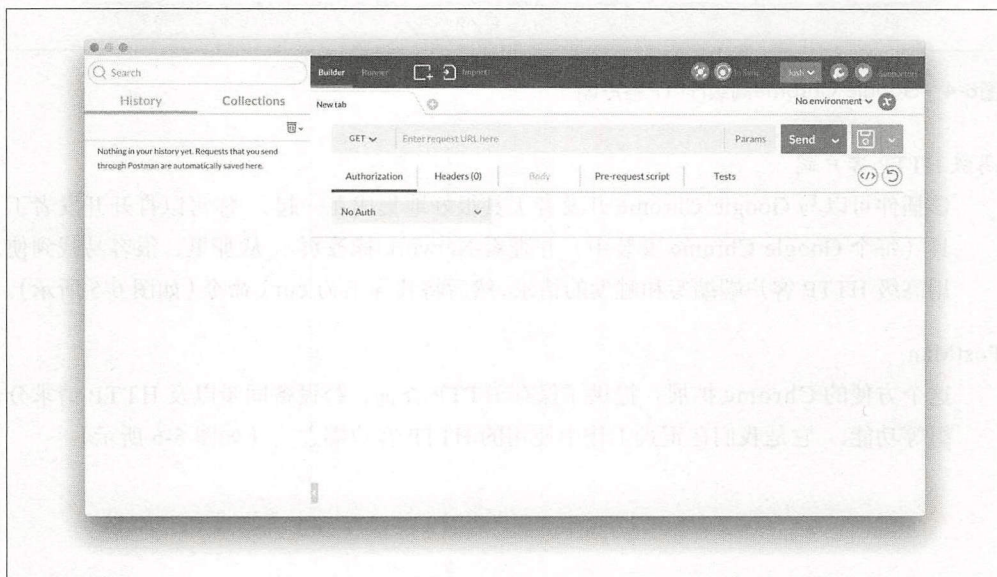


图6-6 PostMan HTTP客户端

RestTemplate

Spring 的 `RestTemplate` 是一个一站式的 HTTP 客户端，支持使用 `template` 方法，只需要一行代码即可调用 `get`、`put`、`post` 等普通 HTTP 请求。与 Spring MVC 相同，`RestTemplate` 支持 `HttpMessageConverter` 策略的内容协商。它可以将对象转换为有效的 HTTP 请求，并将 HTTP 响应转换为对象。它可以通过插入拦截器来处理横切问题，例如使用 HTTP BASIC 和 OAuth 进行身份认证、GZip 压缩和服务解析。

我们来看一下由 Spring Data REST 支持的 REST API。Spring Data REST 会根据指定的 Spring Data repository，自动创建一个具有 HAL 风格链接的超媒体 API。例如，在我们的示例中，Movie 实体具有一个或多个关联的 Actor 实体。

这里需要一个 `RestTemplate`。Spring Boot 自带了 `RestTemplateBuilder` 类，可以简化 `RestTemplate` 实例的配置工作（如示例 6-20 所示）。

示例6-20 配置一个RestTemplate

```
package actors;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpRequest;
import org.springframework.http.client.ClientHttpRequestExecution;
import org.springframework.http.client.ClientHttpRequestInterceptor;
import org.springframework.web.client.RestTemplate;

@Configuration
public class RestTemplateConfiguration {

    @Bean
    RestTemplate restTemplate() {

        Log log = LogFactory.getLog(getClass());

        ClientHttpRequestInterceptor interceptor = (HttpRequest request, byte[] body,
            ClientHttpRequestExecution execution) -> {
            log.info(String.format("request to URI %s with HTTP verb '%s'",
                request.getURI(), request.getMethod().toString()));
            return execution.execute(request, body);
        };

        return new RestTemplateBuilder() ❶
            .additionalInterceptors(interceptor).build();
    }
}
```

❶ RestTemplateBuilder 提供了一个流式 DSL 来配置拦截器、转换器等。

RestTemplate 是 Spring 框架中一个通用的 HTTP 工具。我们通过示例 6-21 来看看它的功能。

示例6-21 使用RestTemplate

```
package actors;
```

```
import com.fasterxml.jackson.databind.JsonNode;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.Resources;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
```

```
import java.net.URI;
import java.util.Collections;
```

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
```

```
public class RestTemplateTest {
```

```
    private Log log = LogFactory.getLog(getClass());
```

```
    private URI baseUri;
```

```
    private ConfigurableApplicationContext server;
```

```
    private RestTemplate restTemplate;
```

```
    private MovieRepository movieRepository;
```

```
    private URI moviesUri;
```

```
    @Before
```

```
    public void setUp() throws Exception {
```

```
        this.server = new SpringApplicationBuilder()
            .properties(Collections.singletonMap("server.port", "0"))
            .sources(DemoApplication.class).run();
```



```

int port = this.server.getEnvironment().getProperty("local.server.port",
Integer.class, 8080);

this.restTemplate = this.server.getBean(RestTemplate.class);
this.baseUrl = URI.create("http://localhost:" + port + "/");
this.moviesUri = URI.create(this.baseUrl.toString() + "movies");
this.movieRepository = this.server.getBean(MovieRepository.class);
}

@After
public void tearDown() throws Exception {
    if (null != this.server) {
        this.server.close();
    }
}

@Test
public void testRestTemplate() throws Exception {
    ❶
    ResponseEntity<Movie> postMovieResponseEntity = this.restTemplate

        .postForEntity(moviesUri, new Movie("Forest Gump"), Movie.class);
    URI uriOfNewMovie = postMovieResponseEntity.getHeaders().getLocation();
    log.info("the new movie lives at " + uriOfNewMovie);

    ❷
    JsonNode mapForMovieRecord = this.restTemplate.getForObject(uriOfNewMovie,
        JsonNode.class);
    log.info("\t..read as a Map.class: " + mapForMovieRecord);
    assertEquals(mapForMovieRecord.get("title").asText(),
        postMovieResponseEntity.getBody().title);

    ❸
    Movie movieReference = this.restTemplate.getForObject(uriOfNewMovie,
        Movie.class);
    assertEquals(movieReference.title, postMovieResponseEntity.getBody().title);
    log.info("\t..read as a Movie.class: " + movieReference);

    ❹
    ResponseEntity<Movie> movieResponseEntity = this.restTemplate.getForEntity(
        uriOfNewMovie, Movie.class);
    assertEquals(movieResponseEntity.getStatusCode(), HttpStatus.OK);
    assertEquals(movieResponseEntity.getHeaders().getContentType(),
        MediaType.parseMediaType("application/json;charset=UTF-8"));
    log.info("\t..read as a ResponseEntity<Movie>: " + movieResponseEntity);

    ❺

    //@formatter:off
    ParameterizedTypeReference<Resources<Movie>> movies =
        new ParameterizedTypeReference<Resources<Movie>>() {};
    //@formatter:on

```

```

ResponseBody<Resources<Movie>> moviesResponseBody = this.restTemplate
    .exchange(this.moviesUri, HttpMethod.GET, null, movies);
Resources<Movie> movieResources = moviesResponseBody.getBody();
movieResources.forEach(this.log::info);
assertEquals(movieResources.getContent().size(), this.movieRepository.count());
assertTrue(movieResources.getLinks().stream()
    .filter(m -> m.getRel().equals("self")).count() == 1);
}
}

```

- ❶ RestTemplate 提供了所有你需要的工具方法，支持常见的 HTTP 动词，例如 POST。
- ❷ 你可以要求 RestTemplate 将响应转换为适当形式后再返回。在这里，我们需要一个 Jackson 框架的 JsonNode 返回值。Json 节点允许解析 JSON，就像 XML 的 DOM 节点一样。
- ❸ 在可能的情况下，该转换也适用于域对象。在这里，我们需要将 JSON 映射（也使用 Jackson 框架）到一个 Movie 实体。
- ❹ ResponseEntity<T> 的返回值封装了几部分内容，包括已转换的响应数据、HTTP 响应头和状态码信息。
- ❺ REST API 会使用 Spring HATEOAS 的 Resources<T> 对象来封装 Resource<T> 实例集合，依次将各个数据包装起来，并提供相应的链接。在前面的例子中，我们返回了一个 .class 字面值，但是 Java 没有办法只通过字面值就获得像 Resource<Movie>.class 这样的东西，因为相关信息已经在编译时通过类型擦除被删掉了。保留泛型参数信息的唯一方法是通过子类化形成一个层级结构。Spring 的 ParameterizedTypeReference 可以支持这种被称为类型记号（*type token*）的模式。ParameterizedTypeReference 是一个抽象类，所以必须被子类化。在这里，ptr 看起来像是一个实例变量，但实际上它是一个匿名的子类，它可以在运行时用 Resources<Movie> 来回答“你的泛型参数是什么？”的问题。通过这种方式，RestTemplate 可以正确绑定返回的 JSON 值。

我们的超媒体 API 使用了 HAL 协议，HAL 也会发布相关资源的链接。HAL 风格的链接，因为一个链接跟在另一个链接之后，所以会形成某种链式结构。解析一个具体的资源，实际上就是遵循这个链条，从一个链接跳转到另一个链接，直到最后到达相关资源。这种方式很直接，但是代码很冗余。如果我们从 / 开始，获取一个位于 /actors/search/by-movie? movie=Cars 的资源，那么至少需要三步：

- 获取根资源 /，并找到 search 链接。
- 找到查询电影标题的搜索链接端点 by-movie。
- 实际运行搜索，传入 movie 请求参数 Cars。

Spring HATEOAS 包含了一个方便的 Traverson 客户端,它可以接受一个链接路径(path),并根据路径进行递归搜索,直到产生最终结果(如示例 6-22 所示)。



Spring Java 客户端的灵感来源于同名的 JavaScript 库 (<https://github.com/bastii302/traverson>), 所以在浏览器客户端中也可以使用这个功能!

示例6-22 使用RestTemplate来配置Traverson客户端

```
package actors;

import
org.springframework.boot.context.embedded.EmbeddedServletContainer
InitializedEvent;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.event.EventListener;
import org.springframework.hateoas.MediaTypes;
import org.springframework.hateoas.client.Traverson;
import org.springframework.web.client.RestTemplate;

import java.net.URI;

@Configuration
public class TraversonConfiguration {

    private int port;

    private URI baseUri;

    //@formatter:off
    @EventListener
    public void embeddedPortAvailable(
        EmbeddedServletContainerInitializedEvent e) {
        this.port = e.getEmbeddedServletContainer().getPort();
        this.baseUri = URI.create("http://localhost:" + this.port + '/');
    }
    //@formatter:on

    ①
    @Bean
    @Lazy
    Traverson traverson(RestTemplate restTemplate) {
        Traverson traverson = new Traverson(this.baseUri, MediaTypes.HAL_JSON);
        traverson.setRestOperations(restTemplate);
        return traverson;
    }
}
```

- ❶ 使用基础 URI、一个希望客户端处理的 MediaType 示例，以及一个代理底层 HTTP 调用的 RestTemplate 实例，来配置一个 Traverson 客户端。

指定一组链接作为搜索路径,然后使用 Traverson 按照该路径进行搜索(如示例 6-23 所示)。

示例6-23 通过RestTemplate来使用Traverson客户端

```
package actors;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.Resources;
import org.springframework.hateoas.client.Traverson;

import java.util.Collections;
import java.util.stream.Collectors;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

public class TraversonTest {

    private Log log = LogFactory.getLog(getClass());

    private ConfigurableApplicationContext server;

    private Traverson traverson;

    @Before
    public void setUp() throws Exception {

        this.server = new SpringApplicationBuilder()
            .properties(Collections.singletonMap("server.port", "0"))
            .sources(DemoApplication.class).run();
        // this.server =
        // SpringApplication.run(DemoApplication.class);
        this.traverson = this.server.getBean(Traverson.class);
    }

    @After
    public void tearDown() throws Exception {
        if (null != this.server) {
            this.server.close();
        }
    }
}
```



```

@Test
public void testTraverson() throws Exception {

    String nameOfMovie = "Cars";

    ❶
    Resources<Actor> actorResources = this.traverson
        .follow("actors", "search", "by-movie")
        .withTemplateParameters(Collections.singletonMap("movie", nameOfMovie))
        .toObject(new ParameterizedTypeReference<Resources<Actor>>() {
        });

    actorResources.forEach(this.log::info);
    assertTrue(actorResources.getContent().size() > 0);
    assertEquals(
        actorResources.getContent().stream()
            .filter(actor -> actor.fullName.equals("Owen Wilson"))
            .collect(Collectors.toList()).size(), 1);
}
}

```

- ❶ `Traverson#follow` 方法简化了遍历多个链接的工作，并能够根据需要替换 URI 参数模板的相应部分。`Traverson#follow` 方法也接受 `ParameterizedTypeReference<T>` 参数，就像 `RestTemplate` 一样。

`RestTemplate` 是 Spring 构建 REST 服务的核心部分。在 Spring Framework 5 中，有一个被称为 `WebClient` 的完全响应式客户端，对于那些长时间的输入 / 输出处理过程，它可以很好地替代 `RestTemplate`。`RestTemplate` 支持 Spring Cloud Security 中的 OAuth 令牌，Spring Cloud 中的服务解析以及与其他 Spring 组件的集成。

总结

在本章中，介绍了如何使用 Spring Boot 构建有效的 REST API。REST 是在开放的、多平台、多语言网络上构建可伸缩服务的一种方法。REST 也涉及其他问题，例如安全、冗余和缓存。虽然我们从 Spring 的角度讨论了 REST 的原理，但是它适用于任何语言和平台。

第 7 章

路由

云原生系统是动态的，服务也会随着需求不断增加或减少。对于服务来说，即使其中一个实例消失或者有新的实例添加进来，都必须能够发现其他服务并进行通信。我们不能依靠固定的 IP 地址，IP 地址会将客户端与服务耦合到一起。我们需要一些其他的手段，能够灵活、动态地重新定义路由。

我们可以使用 DNS，但 DNS 不适合在云环境中使用。DNS 的好处是客户端可以使用 DNS 缓存。但是缓存意味着客户端可能会“解析”到已经不再存在的 IP 地址。你可以用较低的生存时间（TTL）值使这些缓存失效（快速），但是这样必然会导致花费大量时间来重新解析 DNS 记录。在云环境中，DNS 需要额外的解析时间，因为请求必须离开云环境，然后通过路由器重新进入。许多云服务商支持多宿主 DNS 解析，包括提供一个私有地址和公开地址。在这种情况下，在云中服务之间的调用是非常快速和稳定的，并且实际上是免费的（比如带宽等成本），但是这需要代码能够处理复杂的 DNS 情况。这种实现上的复杂性，也会给开发者在本地环境中的重现带来困难。

DNS 也是一个相当简单的协议。它不能回答有关系统状态或拓扑结构这类基本问题。假设你有一个 REST 客户端，根据 DNS 来调用在某个节点上运行的服务。如果服务实例关闭，那么我们的客户端会被一直阻塞没有响应。我们希望每个人都采用了迈克尔·尼加德（Michael Nygard）那本令人惊叹的书 *Release It!* (Pragmatic Programmers) 中的建议，主动设置了客户端的超时时间。但是很可惜，大多数人不会这样做。（提个小问题，在 Java 中 `java.net.URLConnection` 的默认超时时间是多少？你确定已经在所有的代码中设置了适当的客户端超时时间了吗？是你主动设置的吗？）

当你处理 DNS 负载均衡时，路由会变得特别重要。负载均衡器是你需要管理的另一个运维基础架构（或者一般由其他团队来管理），但是其有一定的局限性。通常，负载均衡器可以很好地实现轮询算法，而且大多数甚至可以处理“黏性会话”，即通过请求头

(比如 Java 的 JSESSIONID) 将客户端的请求固定在指定节点上。但是负载均衡器不能实现一些更复杂的需求。如果你想要根据 JSESSIONID 以外的信息, 例如 OAuth 访问令牌或 JSON Web 令牌 (JWT), 将某个客户端的请求绑定到指定节点上, 该如何做到? 又或许你正在做一些有状态的事情, 比如在某个节点上进行流式视频传输, 但是这其中没有 HTTP 会话的概念, 所以实际上你无法进行负载均衡。

显然, DNS 负载均衡池中的所有节点也必须是可路由的。有时我们不希望服务通过 DNS 进行路由! 当然, 不确定的安全就是没有安全, 但是我们应该尽可能减少对外暴露的接口。事实上, 即使我们让客户端直接连接到节点, 也必须确保节点在发生故障时能够正确地退出负载均衡。否则的话, 客户端将被路由到一个无法处理请求的节点。并不是所有的负载均衡器都能够做到这一点。某些负载均衡器能够查询服务实例的健康状况, 并根据响应的状态码从负载均衡池中逐出节点。(你也可以使用 Spring Boot 的 Actuator 端点 /health 实现同样的效果)。有些平台 (例如 JVM) 的库会缓存 DNS 第一个解析的 IP 地址, 并在随后的连接中重新使用它。从大的方面来说, 这是一个好主意, 但它会让 DNS 负载均衡失去效果。

你可以将虚拟的负载均衡器作为某种代理来解决其中一些局限性, 但它们仍然无法解决较大的问题。所有集中式的负载均衡器都不知道系统的状态和工作负载。即使是一个理想的轮询式 DNS 负载均衡器, 也会将初始连接分配给不同的服务, 而不是根据工作的轻重情况。并不是所有的请求都是平等的, 一些客户端会比其他客户端消耗更多的资源, 并可能需要更长的时间来处理。如果没有合适的智能化负载均衡策略, 这可能会导致某些节点过载, 而其他节点闲置。

总之, 路由 (通常) 是一个编程范围内的话题, DNS 并不适合。

DiscoveryClient 接口

还有其它方法可以代替集中式的负载均衡方法。我们希望获得某个服务 ID 与可用服务主机和端口之间的逻辑映射。这里很适合使用服务注册中心。服务注册中心的主要缺点是它们会侵入到应用程序代码中。你的代码必须知道并使用注册中心。对于无法使用服务注册中心的客户端, 有些服务注册中心 (例如 Hashicorp Consul) 可以作为它们的 DNS 服务器。服务注册中心最终就像是云上的一个电话簿一样。它包含所有服务实例的一个列表, 并提供了一个 API。一些服务注册中心会更加复杂, 但是它们至少都支持, 发现哪些服务可用, 以及指出这些服务实例的位置。

并非所有的服务注册都是平等的。它们也受到物理和 CAP 定理的限制。CAP 指出, 分布式系统不可能同时提供以下三个属性: 一致性、可用性和分区容忍性。服务注册中心

的作用是保持对系统一致的可见性。你需要评估各个服务注册中心在 CAP 定理中所处的位置。你还需要评估服务注册中心是否在核心功能之外还提供其他的功能。它是否支持安全和加密？它是否可以作为一个键/值存储来进行集中式配置，还是像第 3 章中关于配置的介绍一样，应该使用 Spring Cloud Config Server？

Spring Cloud 提供了 `DiscoveryClient` 接口，使得客户端可以轻松地使用不同类型的服务注册中心。Spring Cloud `DiscoveryClient` 使得我们可以轻松替换不同的实现。Spring Cloud 将 `DiscoveryClient` 插入到其技术栈的各个部分中，使其几乎对客户端是透明的。在写本书时，Cloud Foundry、Apache Zookeeper、Hashicorp Consul 和 Netflix Eureka 都已经有了可以在生产环境中使用的 `DiscoveryClient` 实现了，而 ETCD 的实现暂时还不适合用在生产环境。`DiscoveryClient` 接口非常简单，你可以让 Spring Cloud 与其他服务注册中心一起工作。从概念上讲，`DiscoveryClient` 是只读的，如示例 7-1 所示。

示例7-1 `DiscoveryClient`接口

```
package org.springframework.cloud.client.discovery;

import java.util.List;
import org.springframework.cloud.client.ServiceInstance;

public interface DiscoveryClient {
    String description();

    ServiceInstance getLocalServiceInstance();

    List<ServiceInstance> getInstances(String var1);

    List<String> getServices();
}
```

某些服务注册中心需要客户端向注册中心进行注册。各种 `DiscoveryClient` 实现可以在应用程序启动时帮你来执行此类操作。Cloud Foundry 的 `DiscoveryClient` 实现不需要客户端向注册中心注册，因为 Cloud Foundry 已经知道服务位于哪个主机和端口上，它把服务摆在首要地位！

服务注册有很多很好的选择。因为 Spring 提供的是一个接口，所以并不限制对它的选择。稍后我们可以轻松切换到另一个实现。本章我们先来了解一下 Netflix Eureka。Netflix Eureka 多年来一直为 Netflix 服务。Eureka 可以完全使用 Spring Boot 来安装和运行！



虽然 Netflix Eureka 几乎可以任何事情，但你不应该这样做。相反，你应该使用 Pivotal Cloud Foundry 或 Pivotal Web Services 上基于 Spring Cloud Services 的 Netflix，它们在安全性、可扩展性和冗余性方面已经做了很多工作。

为了搭建 Eureka 服务注册中心，需要在 Spring Boot 项目中指定 `org.springframework.cloud:spring-cloud-starter-eureka-server` 依赖，然后通过 `@EnableEurekaServer` 进行初始化，如示例 7-2 所示。

示例7-2 初始化一个Eureka服务注册中心

```
package demo;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
```

❶

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServiceApplication {

    public static void main(String args[]) {
        SpringApplication.run(EurekaServiceApplication.class, args);
    }
}
```

❶ `@EnableEurekaServer` 注解会为 Eureka 服务注册中心初始化一个实例。



这不是一个可用在生产环境中的配置！为了演示，我们省略了很多其他的事情。但是在生产环境中，你还需要关心这些事情。也许 Netflix Eureka 可以满足你的需求，但这并不意味着搭建集群（<https://github.com/spring-cloud/spring-cloud-netflix/issues/203>）这样的事情很容易！希望你使用一个平台（像 Cloud Foundry 一样）来自动化运维这些事情。

你需要配置服务注册中心。通常我们会在端口 8761 上启动服务，并且不希望注册中心向其他节点注册自己（如示例 7-3 所示）。

示例7-3 配置一个简单的Eureka服务注册中心

```
server.port=${PORT:8761}
```

❶

```
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

❷

```
eureka.server.enable-self-preservation=false
```

- ① 我们不希望 Eureka 试图注册自己。
- ② 如果大部分已注册的实例在一段时间内停止发送心跳信号, Eureka 会认为这是由于网络原因造成的, 不会注销没有响应的服务。这是 Eureka 的自我保护模式。通常把这个选项设置为 `true` 可能会好一些, 但是这也意味着如果你有一小组实例 (就像你会在本地机器上启动几个节点来实验一样), 那么当某个实例被注销时你也不会发现。

我们来看一下图 7-1 所示新的 Eureka 服务注册中心! 如果你把鼠标放在 Spring 的叶子图标上, 会看到它的动画效果。

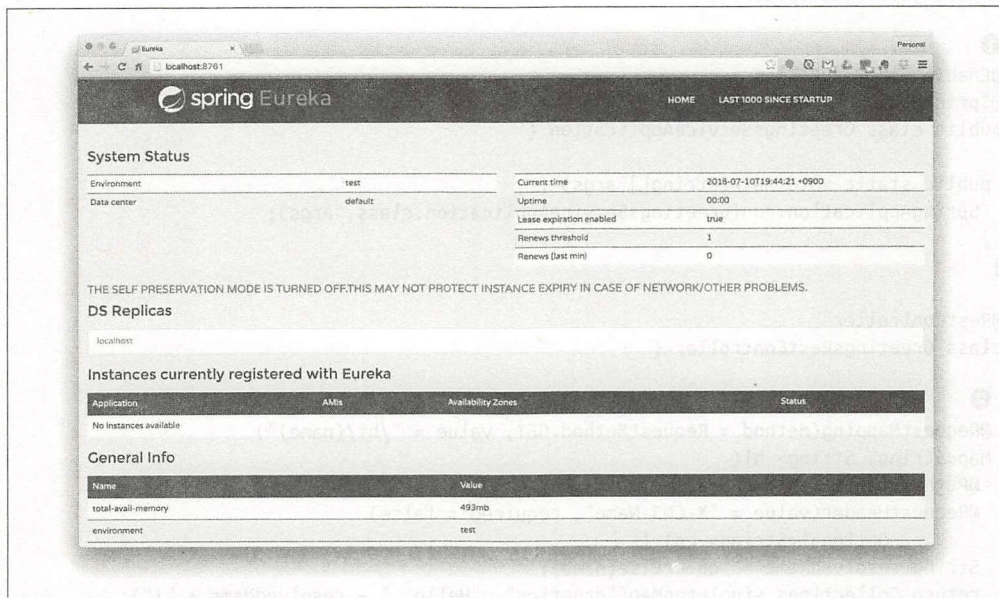


图7-1 默认初始化的Eureka服务注册中心

现在, 我们的注册中心已经可以使用了。它不是高可用的, 所以不能用于生产环境。事实上, Eureka 偶尔会在控制台上提示红色的错误消息, 提醒我们需要配置复制节点, 否则我们的注册中心会非常脆弱。记住, Eureka 将成为 REST 服务的神经系统: 其他服务将通过 Eureka 彼此进行交流。

但是它是可以使用的。让我们以一个客户端和一个服务为例, 将它们都注册到注册中心。首先创建一个简单的 REST API, 稍后使用。这个应用程序依赖于 `org.springframework.cloud:spring-cloud-starter-eureka` (提供了支持 Netflix Eureka 的 Spring Cloud DiscoveryClient 实现) 和 `org.springframework.boot:spring-boot-starter-web` (如示例 7-4 所示)。

示例7-4 一个会返回问候语的简单REST API

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.util.StringUtils;
import org.springframework.web.bind.annotation.*;
import javax.servlet.http.HttpServletRequest;
import java.util.Collections;
import java.util.Map;
import java.util.Optional;

1
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingsServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(GreetingsServiceApplication.class, args);
    }
}

@RestController
class GreetingsRestController {

2
    @RequestMapping(method = RequestMethod.GET, value = "/hi/{name}")
    Map<String, String> hi(
        @PathVariable String name,
        @RequestHeader(value = "X-CNJ-Name", required = false)
            Optional<String> cn) {
        String resolvedName = cn.orElse(name);
        return Collections.singletonMap("greeting", "Hello, " + resolvedName + "!");
    }
}
```

❶ @EnableDiscoveryClient 会激活 Spring Cloud DiscoveryClient 接口。

❷ 它会返回一个带有 greeting 属性的 JSON 响应。

我们的服务只需要识别出自己，并配置如何与 Eureka 进行交互。Spring Cloud 会像 Spring Boot 一样，从 applicaiton.{yaml,properties} 中找到需要的值，但是它还需要一些比应用程序生命周期更早的值。它希望在 boot strap.properties 中找到 Spring Cloud Config Server 的地址等配置项，如示例 7-5 所示。

示例7-5 bootstrap.properties

```
spring.application.name=greetings-service ❶  
  
server.port=${PORT:0} ❷  
  
eureka.instance.instance-id=\ ❸  
    ${spring.application.name}:${spring.application.instance_id:${random.value}}
```

- ❶ 该应用程序将被注册为 `greetings-service`。
- ❷ 它将在一个随机的端口启动。
- ❸ 你希望每个注册的服务都有独特的 ID 吗？Spring Cloud 会为你提供一个有用的、跟节点有关的默认值。这里，已经覆盖了默认的注册 ID，以保证它们是唯一的。

启动几个 `greetings-service` 的实例，然后刷新 Eureka 服务注册中心，你会看到新注册的实例。它们现在已经将状态公开，供其他的服务使用（如图 7-2 所示）。

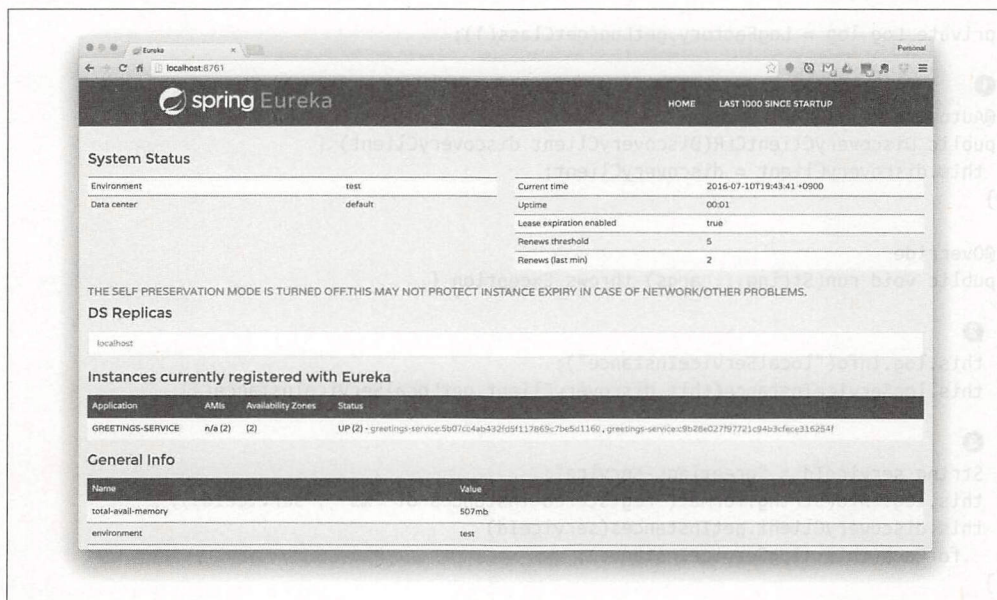


图7-2 已经含有注册实例的Eureka!

我们假设有一个简单的客户端，例如一个边缘服务。我们将在第 8 章中讨论更多有关边缘服务的内容，但现在暂时认为边缘服务是请求进入系统的第一个入口。边缘服务将作为 `greetings-service` 的客户端，它将通过注册中心来解析服务地址，然后与 `greetings-service` 服务进行交互。

我们的新客户端也使用了 `spring-cloud-starter-eureka`, 并使用 `@EnableDiscoveryClient` 注解标注。也可以直接使用 `DiscoveryClient` 接口来询问已注册的服务实例（如示例 7-6 所示）。

示例 7-6 `DiscoveryClientCLR.java`

```
package com.example;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.stereotype.Component;

@Component
public class DiscoveryClientCLR implements CommandLineRunner {

    private final DiscoveryClient discoveryClient;

    private Log log = LogFactory.getLog(getClass());

    ❶
    @Autowired
    public DiscoveryClientCLR(DiscoveryClient discoveryClient) {
        this.discoveryClient = discoveryClient;
    }

    @Override
    public void run(String... args) throws Exception {

        ❷
        this.log.info("localServiceInstance");
        this.logServiceInstance(this.discoveryClient.getLocalServiceInstance());

        ❸
        String serviceId = "greetings-service";
        this.log.info(String.format("registered instances of '%s'", serviceId));
        this.discoveryClient.getInstances(serviceId)
            .forEach(this::logServiceInstance);
    }

    private void logServiceInstance(ServiceInstance si) {
        String msg = String.format("host = %s, port = %s, service ID = %s",
            si.getHost(), si.getPort(), si.getServiceId());
        log.info(msg);
    }
}
```

❶ 注入 Spring Cloud 已经配置的 `DiscoveryClient` 实现。

- ② 获取有关当前实例的信息，即当前运行的应用程序会在 Eureka 中注册哪些信息？
- ③ 查找并遍历所有已注册的 greetings-service 实例。

通过 `DiscoveryClient` 可以很容易使用注册中心，遍历所有已注册的实例。如果注册的服务有多个实例，则需要选择一个特定的版本。我们可以从返回的实例中随机选择，如同轮询式的负载均衡。我们可能还希望利用一下应用程序和实例的响应能力，实现诸如加权响应式的负载均衡。无论是什么策略，我们只需要在代码中描述一次，然后在任何调用服务的时候重用它。这就是客户端负载均衡的本质。

Netflix Ribbon 是一个客户端负载均衡库。它支持不同的负载均衡策略，包括轮询和加权响应式的负载均衡，但它的优点在于可扩展性。我们来看示例 7-7 中的初级示例。

示例7-7 RibbonCLR.java

```
package com.example;

import com.netflix.loadbalancer.*;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.stereotype.Component;

import java.net.URI;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

@Component
public class RibbonCLR implements CommandLineRunner {

    private final DiscoveryClient discoveryClient;

    private final Log log = LogFactory.getLog(getClass());

    @Autowired
    public RibbonCLR(DiscoveryClient discoveryClient) {
        this.discoveryClient = discoveryClient;
    }

    @Override
    public void run(String... args) throws Exception {

        String serviceId = "greetings-service";
```



```

1
List<Server> servers = this.discoveryClient.getInstances(serviceId).stream()
    .map(si -> new Server(si.getHost(), si.getPort()))
    .collect(Collectors.toList());

2
IRule roundRobinRule = new RoundRobinRule();

BaseLoadBalancer loadBalancer = LoadBalancerBuilder.newBuilder()
    .withRule(roundRobinRule).buildFixedServerListLoadBalancer(servers);

IntStream.range(0, 10).forEach(i -> {
3
    Server server = loadBalancer.chooseServer();
    URI uri = URI.create("http://" + server.getHost() + ":" + server.getPort()
        + "/"");
    log.info("resolved service " + uri.toString());
});
}
}

```

- ❶ 注入 Spring Cloud 已经配置的 `DiscoveryClient` 实现。
- ❷ 获取有关当前实例的信息，即当前运行的应用程序会在 Eureka 中注册哪些信息？
- ❸ 查找并遍历所有已注册的 `greetings-service` 实例。

这个示例可以工作，但是很乏味！值得庆幸的是，Spring Cloud 自动在框架的各个层次都集成了客户端负载均衡。其中一个尤其有效的例子就是对 Spring `RestTemplate` 的自动配置。`RestTemplate` 支持在 HTTP 请求处理前后放置拦截器。我们可以使用 Spring Cloud 的 `@LoadBalanced` 拦截器，在 `RestTemplate` 上配置一个支持 Ribbon 的负载均衡拦截器（如示例 7-8 所示）。

示例7-8 `LoadBalancedRestTemplateConfiguration.java`

```

package com.example;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class LoadBalancedRestTemplateConfiguration {

1
    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

```

- ❶ @LoadBalanced 注解是一个限定符注解。Spring 使用限定符来消除有歧义的 bean 定义，并指定处理某些 bean。在这里，我们标记该 RestTemplate 实例需要配置一个负载均衡拦截器。

如此以后，我们的工作就变得简单很多！负载均衡器会提取任何 HTTP 请求的 URI，并将主机视为 DiscoveryClient（在本例中为 Netflix Eureka）要解析的服务 ID，同时使用 Netflix Ribbon 进行负载均衡（如示例 7-9 所示）。

示例7-9 LoadBalancedRestTemplateCLR.java

```
package com.example;

import com.fasterxml.jackson.databind.JsonNode;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.util.Collections;
import java.util.Map;

@Component
public class LoadBalancedRestTemplateCLR implements CommandLineRunner {

    private final RestTemplate restTemplate;

    private final Log log = LogFactory.getLog(getClass());

    ❶
    @Autowired
    public LoadBalancedRestTemplateCLR(@LoadBalanced RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @Override
    public void run(String... strings) throws Exception {

        Map<String, String> variables = Collections.singletonMap("name",
            "Cloud Natives!");

        ❷
        ResponseEntity<JsonNode> response = this.restTemplate.getForEntity(
            "//greetings-service/hi/{name}", JsonNode.class, variables);
        JsonNode body = response.getBody();
        String greeting = body.get("greeting").asText();
        log.info("greeting: " + greeting);
    }
}
```


- ① 我们在 bean 的生产者上和注入点，都使用相同的 `@LoadBalanced` 限定符注解。
- ② 这里使用 `RestTemplate`，因为我们可能会为主机指定一个服务 ID (`greetings-service`)，来代替 DNS 地址。

Cloud Foundry Route 服务

客户端负载均衡为我们提供了很多路由决策的能力，作为开发者，我们可以更简单地在代码中实现路由逻辑。对于我们的服务系统来说，路由是保留在本地的，所以如果我们希望改变它，可以不用担心影响其他人使用服务。不过，期望所有第三方客户端都使用 Netflix Ribbon，并以相同的方式使用 API 是不现实的，尤其是对于响应客户端请求的边缘服务来说，可能会接收到许多来自不同设备和不同格式的请求。例如，iOS 客户端不会希望使用 Apache Zookeeper 和 Netflix Ribbon 来完成它们的工作。不过，对于那些处理请求、处理路由，然后将请求转发给下游服务的中间服务来说，客户端负载均衡可能是有用的。这种组件也适用于所有类型的服务，无论选择用什么语言实现。

Cloud Foundry 支持一种特殊的组件（一种路由服务），它不仅为我们带来了客户端负载均衡的大多数好处，并且具有一个能够控制路由行为的中心化组件。路由服务是 Cloud Foundry 中的另一个扩展项，你可以开发构建包、应用程序和服务代理，并将它们插入你的平台。路由服务最终是一个 HTTP 服务，它会接收发给指定主机和域的所有 HTTP 请求，然后对它们进行同样的处理、转换或转发。

路由服务（理论上）是为了作为任意应用程序在请求链中插入的通用代理，所以采取了一些约束。它们必须同时支持 HTTP 和 HTTPS。Cloud Foundry 目前不支持将一个请求链接到另一个请求的路由服务。Cloud Foundry 将路由服务视为另一种由用户提供的服务。它与我们在本书中讨论的其他由用户提供的服务有一点不同，路由服务和应用程序之间的连接是根据应用程序的主机名和域来指定的，而不是根据应用程序的名称。

我们来看一个例子。由于我们需要接收并传递所有 HTTP 或 HTTPS 请求，所以我们对 Spring 的 `RestTemplate` 进行配置，让它信任所有内容并忽略错误（如示例 7-10 所示）。

示例 7-10 `RouteServiceApplication.java`

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.http.client.ClientHttpResponse;
import org.springframework.http.client.SimpleClientHttpRequestFactory;
import org.springframework.web.client.DefaultResponseErrorHandler;
```

```

import org.springframework.web.client.RestTemplate;

import javax.net.ssl.HttpURLConnection;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.Proxy;
import java.net.URL;
import java.security.KeyManagementException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;

/**
 * @author Ben Hale
 */
@SpringBootApplication
public class RouteServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(RouteServiceApplication.class, args);
    }

    ❶
    @Bean
    RestTemplate restOperations() {
        RestTemplate restTemplate = new RestTemplate(
            new TrustEverythingClientHttpRequestFactory()); ❷
        restTemplate.setErrorHandler(new NoErrorsResponseErrorHandler()); ❸
        return restTemplate;
    }

    private static class NoErrorsResponseErrorHandler extends
        DefaultResponseErrorHandler {

        @Override
        public boolean hasError(ClientHttpResponse response) throws IOException {
            return false;
        }
    }

    private static final class TrustEverythingClientHttpRequestFactory extends
        SimpleClientHttpRequestFactory {

        private static SSLContext getSslContext(TrustManager trustManager) {
            try {
                SSLContext sslContext = SSLContext.getInstance("SSL");
                sslContext.init(null, new TrustManager[] { trustManager }, null);
                return sslContext;
            }

```



```

catch (KeyManagementException | NoSuchAlgorithmException e) {
    throw new RuntimeException(e);
}
}

@Override
protected HttpURLConnection openConnection(URL url, Proxy proxy)
    throws IOException {
    HttpURLConnection connection = super.openConnection(url, proxy);
    if (connection instanceof HttpsURLConnection) {
        HttpsURLConnection httpsConnection = (HttpsURLConnection) connection;
        SSLContext sslContext = getSslContext(new TrustEverythingTrustManager());
        httpsConnection.setSSLSocketFactory(sslContext.getSocketFactory());
        httpsConnection.setHostnameVerifier((s, session) -> true);
    }
    return connection;
}
}

private static final class TrustEverythingTrustManager implements
    X509TrustManager {

    @Override
    public void checkClientTrusted(X509Certificate[] x509Certificates, String s)
        throws CertificateException {
    }

    @Override
    public void checkServerTrusted(X509Certificate[] x509Certificates, String s)
        throws CertificateException {
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return new X509Certificate[0];
    }
}
}

```

- ❶ 配置一个 RestTemplate。
- ❷ 信任一切请求。
- ❸ 并忽略错误。

此路由服务会在接收所有请求的前后，将请求记录到日志中（如示例 7-11 所示）。

示例7-11 Controller.java

```
package com.example;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestOperations;
import org.springframework.web.client.RestTemplate;

import java.net.URI;
import java.util.Arrays;
import java.util.Collections;

/**
 * @author Ben Hale
 */
@RestController
class Controller {

    ①
    private static final String FORWARDED_URL = "X-CF-Forwarded-Url";

    private static final String PROXY_METADATA = "X-CF-Proxy-Metadata";

    private static final String PROXY_SIGNATURE = "X-CF-Proxy-Signature";

    private final Log logger = LogFactory.getLog(this.getClass());

    private final RestOperations restOperations;

    @Autowired
    Controller(RestTemplate restOperations) {
        this.restOperations = restOperations;
    }

    ②
    @RequestMapping(headers = { FORWARDED_URL, PROXY_METADATA, PROXY_SIGNATURE })
    ResponseEntity<?> service(RequestEntity<byte[]> incoming) {

        this.logger.info("incoming request: " + incoming);

        HttpHeaders headers = new HttpHeaders();
        headers.putAll(incoming.getHeaders());
        headers.put("X-CNJ-Name", Collections.singletonList("Cloud Natives"));
```



```

3
URI uri = headers
    .remove(FORWARDED_URL)
    .stream()
    .findFirst()
    .map(URI::create)
    .orElseThrow(
        () -> new IllegalStateException(String.format("No %s header present",
            FORWARDED_URL)));

4
RequestEntity<?> outgoing = new RequestEntity<>(
    ((RequestEntity<?>) incoming).getBody(), headers, incoming.getMethod(), uri);

this.logger.info("outgoing request: {}" + outgoing);

return this.restOperations.exchange(outgoing, byte[].class);
}
}

```

- ❶ Cloud Foundry 会传递所有为请求提供上下文的自定义头信息。
- ❷ 将这些头信息作为一个选择器，来确定路由服务是否应该处理请求。
- ❸ 在处理请求的方法中，通过提取合适的头信息来确定最终的请求。
- ❹ 将请求转变为一个可以被 RestTemplate 处理的输出请求。

这是一个很简单的例子，但是延伸的可能性很多。我们可以对请求进行预处理，认证身份信息，或者采用其他身份认证方式。可以插入一些限速的逻辑，或者通过客户端负载均衡以某种方式转发请求。可以测算进入系统的请求数量。可以简单地将其作为一种通用的服务级广告，并通过全局状态或请求上下文（也许是用户的点击流信息？）来丰富请求的内容。

在这里你可能想要做很多的事情，包括计数、速率限制、认证等常见事务，你不需要为它们编写代码。Apigee 等 API 网关产品可以很好地解决许多此类问题，并且还提供了一个可配置的 Cloud Foundry 路由服务。Apigee API 网关会连接到你的 Apigee 账户，并且你在 Apigee 上集中配置的所有策略，都会自动应用到你的请求上。

使用 Cloud Foundry 的路由服务相当简单，但与使用任何其他路由服务不完全相同。路由服务最终只是你部署的一个 Cloud Foundry 应用程序。假设你已经为 Cloud Foundry 部署了两个应用程序：route-service（刚刚演示的记录日志的路由服务）以及一个常规的 Spring Boot 和 Spring MVC 应用程序（downstream-service）。在示例 7-12 中，将一个应用程序（downstream-service）连接到路由服务（route-service），以便访问部署

在 Pivotal Web Services 上、默认域为 `cfapps.io` 的应用程序。（如果你使用在其他地方已经部署的 Cloud Foundry 实例，则情况可能会不同。）

示例7-12 将应用程序连接到一个路由服务

```
cf create-user-provided-service route-service -r https://my-route-service.cfapps.io
```

❶

```
cf bind-route-service cfapps.io route-service -hostname my-downstream-service
```

❷

- ❶ 创建一个由用户提供的路由服务，将其指向我们已部署的 `route-service` 的 URL。
- ❷ 将该路由服务绑定到路由为 `http://my-downstream-service.cfapps.io` 的任一应用程序上。

现在，在浏览器中访问 `downstream-service` 几次，然后查看路由服务的日志（`cf logs --recent route-service`）。可以看到记录请求的日志。

总结

对于通过 HTTP 部署的服务（但是并不一定必须这样），路由是一个非常重要的问题，尤其当服务的生命周期是动态的，并且服务拓扑发生变化时，这变得更加重要。我们希望，对于从外部访问系统的同类客户端，在一个固定的、众所周知的地方提供服务，但是同时又要完全保证服务内部之间通信的灵活性。我们可以利用本章所介绍的方法，通过一些调整来解决这些看起来冲突的问题。

边缘服务

微服务不是存在于真空中。它们最终也要为客户端服务。这些客户端各种各样：HTML5 客户端、Android 客户端、iOS 客户端，PlayStation 或者 XBox、智能电视，以及其他几乎所有拥有 MAC 地址（感谢 ARP，只需要一个 IP 地址）的客户端。真的可以是任何东西。新加坡的道路采集了传感器数据，通过云上的服务来帮助优化交通流量。人类就好像带着与网络连接的器官行走在地球上一样！

客户端有许多不同的能力维度，体现在它们可以使用的数据和服务类型上，例如：

- 某些客户端内存容量或处理能力有限，会影响一个客户端可管理的内容量。
- 某些客户端需要指定内容类型或编码。
- 某些客户端需要不同的文档优化模型，有一些是层级化的，而有一些是平级的。
- 某些客户端的屏幕空间可能需要数据逐步进行加载，而不是一次性加载。
- 某些客户端可以更有效地对文档进行流式或分块式传输。
- 用户交互可能会改变所需的响应。
- 它们可以影响元数据字段、传输方式、交互模型等方面。

为了适应每个新的客户端而改装系统中的每一个微服务，这样的成本是令人望而却步的，而且给系统很快带来无法控制的复杂性。我们的目标应该是保持敏捷，保持对系统的控制。相反，在中间环节处理这些问题的服务，我们称之为边缘服务。边缘服务是请求进入应用程序的一个门户。它们的位置十分关键，无论是跟客户端有关的问题，还是像安全（身份认证和授权）、限速和计数、路由等跨服务的问题（有关路由、过滤、API 转换、客户端适配器 API、服务代理，以及其他问题的讨论，请参考第 7 章中的更多内容），都需要边缘服务来处理。

Greetings 服务

在本章中，我们将着眼于解决边缘服务的问题。我们将通过一个简单的、称为 `greetings-service` 的 REST 服务，以及一个 Eureka 服务注册中心来实现。下面我们开始来搭建这些服务。首先，快速搭建一个 Netflix Eureka 实例，即 `service-registry` 模块。这是一个典型的 Spring Boot 应用程序，如同我们在 Spring Initializr (<http://start.spring.io>) 上创建的一样。对于这个模块，我们添加了 `org.springframework.cloud:spring-cloud-starter-eureka-server` 依赖（如示例 8-1 所示）。

示例8-1 在 `service-registry` 模块中搭建一个简单的 Netflix Eureka 服务器

```
package demo;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
```

❶

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServiceApplication {

    public static void main(String args[]) {
        SpringApplication.run(EurekaServiceApplication.class, args);
    }
}
```

❶ `@EnableEurekaServer` 注解会初始化一个 Eureka 服务中心实例。

当然，注册中心还需要一些最基本的配置（如示例 8-2 所示）。

示例8-2 `service-registry` 模块中的 `application.properties`

```
spring.application.name=eureka-service
```

❶

```
eureka.client.register-with-eureka=false
```

```
eureka.client.fetch-registry=false
```

❷

```
eureka.server.enable-self-preservation=false
```

❶ 告诉 Eureka 不要注册自己。

❷ 如果 Eureka 检测到节点在指定时间阈值内没有响应，确保 Eureka 不会主动地缓存集群信息。这对于开发可能很方便，但是在生产环境中应该禁止该功能。

运行 `service-registry` 模块，它会在端口 8761 上启动。然后，需要一个简单的 REST

API——greetings-service，返回一个包含问候语的响应。除了进行服务注册和发现，同时向其他服务广播自己之外，这个服务通常不会受到注意（如示例 8-3 所示）。

示例8-3 在greetings-service模块中的greetingsServiceApplication.java

```
package greetings;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
```

```
①
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingsServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(GreetingsServiceApplication.class, args);
    }
}
```

① 通过 Spring Cloud DiscoveryClient 接口进行服务注册和发现。

我们将在本章中看到许多相同 REST API 的组合，所以我们会通过 Spring profile 文件，选择性地启用或禁用它们。默认 greetings-service 服务只包括了两个必需的 profile，如示例 8-4 所示。

示例8-4 DefaultGreetingsRestController.java

```
package greetings;

import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collections;
import java.util.Map;

@Profile({ "default", "insecure" })
@RestController
@RequestMapping(method = RequestMethod.GET, value = "/greet/{name}")
class DefaultGreetingsRestController {

    @RequestMapping
    Map<String, String> hi(@PathVariable String name) {
        return Collections.singletonMap("greeting", "Hello, " + name + "!");
    }
}
```

我们将在本章的后面再次回顾这个示例。运行该程序，它将在 Eureka 中注册为 `greetings-service`，并且默认在端口 8081 上启动。

一个简单的边缘服务

下面我们来模拟一个简单的客户端 API——一个名为 `greetings-client` 的边缘服务，它将充当外部世界和下游服务之间的一个中介。这个端点将作为一个访问下游服务 `greetings-service` 的客户端，其他的客户端（例如 HTML5 客户端、iOS 客户端、Android 客户端等）都会与这个服务进行通信。对于客户端来说，API 适配器用来返回一个数据视图，或者将其他服务的数据整合成一个新的数据。下面我们来了解一下各种创建 API 适配器的方式。

新的入门级 API 适配器非常简单：它使用 `Spring Cloud DiscoveryClient` 接口，就像 `greetings-service` 一样，因此其他节点可以通过服务注册和发现来使用 API 适配器。首先，我们需要为适配器指定一个独立的端口和一个 `spring.application.name` 名称。在示例 8-5 中，使用 `greetings-client` 作为应用程序的名称。

示例8-5 `GreetingsClientApplication.java`

```
package greetings;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
```

❶

```
@EnableDiscoveryClient
@SpringBootApplication
public class GreetingsClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(GreetingsClientApplication.class, args);
    }
}
```

❶ 激活服务注册和发现。

我们来看第一个客户端的端点。由于这些端点使用了客户端负载均衡（感谢 Spring Cloud 提供了与 Netflix Ribbon 的集成），使得我们可以通过 `RestTemplate` 向下游服务发出请求。每个请求都由 Spring Cloud 的拦截器进行预处理，并使用 `@LoadBalanced` 限定符注解标注。拦截器会从 URI 中提取主机名，通过 `DiscoveryClient` 发现注册中心中的所有服务实例。被发现的实例会被传递给 Netflix Ribbon 客户端的负载均衡器，然后由负载均衡器决定将请求路由到哪个节点。

调用 REST 服务最终会返回一个 JSON 字符串响应。这里，我们使用 Spring 框架的

ParameterizedTypeReference，它是一个超类型标记模式的实现，用来将返回的 JSON 字符串强制转换为我们可以使用的类型。RestTemplate 会使用 exchange 方法中最后一个类型为 Class<T> 或 ParameterizedTypeReference<T> 的参数，来确定期望返回结果的类型。在这个示例中，我们希望将响应的 JSON 字符串转换为 Map<String,String> 类型（如示例 8-6 所示）。

示例8-6 RestTemplateGreetingsClientApiGateway.java

```
package greetings;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Profile;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.client.RestTemplate;

import java.util.Map;

@Profile({ "default", "insecure" })
@RestController
@RequestMapping("/api")
class RestTemplateGreetingsClientApiGateway {

    private final RestTemplate restTemplate;

    @Autowired
    RestTemplateGreetingsClientApiGateway(
        @LoadBalanced RestTemplate restTemplate) { ❶
        this.restTemplate = restTemplate;
    }

    @GetMapping("/resttemplate/{name}")
    Map<String, String> restTemplate(@PathVariable String name) {

        //@formatter:off
        ParameterizedTypeReference<Map<String, String>> type =
            new ParameterizedTypeReference<Map<String, String>>() {};
        //@formatter:on

        ResponseEntity<Map<String, String>> responseEntity = this.restTemplate
            .exchange("http://greetings-service/greet/{name}", HttpMethod.GET, null,
                type, name);
        return responseEntity.getBody();
    }
}
```

- ❶ @LoadBalanced 是一个限定符注解，它将消费者与指定的 RestTemplate 实例连接起来。为了支持客户端负载均衡，Spring Cloud 会为该 RestTemplate 配置一个拦截器。



超类型标记模式会捕获类信息中的泛型参数，以规避 Java 语言中类型擦除的限制——即 Java 不会在运行时保留实例变量的泛型参数。据我们所知，类型擦除最初是由在 Sun 的 JDK 团队工作的 Neal Gafter (<http://gafter.blogspot.com/2006/12/super-type-tokens.html>) 所提倡的。

RestTemplate 可以方便快捷地调用其他服务。方便是因为我们可以通过 RestTemplate 不断重复调用一个 API，但是，如果要调用多个不同的 API，很快就会变得枯燥乏味。因为我们的边缘服务会调用很多不同的服务，所以我们需要简化客户端的创建工作。

Netflix Feign

我们要简化调用下游服务的工作。一种选择是编写一个 API 客户端，然后重复使用它。它可以与其他团队共享，缩短开发时间。如果你不介意先编写客户端，并且可以让它与服务 API 保持一致，那么这听上去是一个好主意（关于如何确保这种一致性，请参阅第 15 章对于消费者驱动的契约测试的讨论），但是这样的风险在于，部分本该在服务中实现的业务逻辑会放在客户端中实现。虽然我们可以创建自己的客户端，但这是一场艰苦的战役，而且可能会引入更多的错误和不一致性。这时，我们可能会被迫开始尝试使用 RPC 技术来代替 REST。但是，非常不建议这样做，除非你有一个特殊的用例是 REST 无法满足的。否则，你应该考虑使用像 Netflix Feign 这样的技术。

Netflix Feign 是 Netflix 提供的一个库，它使得创建服务客户端就像定义接口和约定一样简单。它可以与 Spring Cloud 很好地集成在一起。我们可以通过在配置类中添加 @EnableFeignClients 注解（或者使用 @SpringBootApplication 注解）并且添加 org.springframework.cloud:spring-cloud-starter-feign 依赖来启用它。以下是一个简单的以接口方式创建客户端的例子。确保使用 feign profile 文件来启动 greetings-client（如示例 8-7 所示）。

示例8-7 GreetingsClient.java

```
package greetings;

import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import java.util.Map;

❶
@EnableFeignClient(serviceId = "greetings-service")
interface GreetingsClient {
```



```

    ②
    @RequestMapping(method = RequestMethod.GET, value = "/greet/{name}")
    Map<String, String> greet(@PathVariable("name") String name); ③
}

```

- ❶ 服务 ID 需要与 DiscoveryClient 接口和 Netflix Ribbon 一起使用，以便实现客户端负载均衡。
- ❷ 通过使用 Spring MVC 请求映射注解，来指定应该调用哪些端点以及如何调用它们。你肯定在服务端（而不是客户端）实现上看到过这样的操作！
- ❸ 返回值用来确定如何对结果序列化。不需要再使用超类型标记模式了！

然后就可以在边缘服务中使用 Feign 客户端了（如示例 8-8 所示）。

示例8-8 FeignGreetingsClientApiGateway.java

```

package greetings;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.*;

import java.util.Map;

    ❶
    @Profile("feign")
    @RestController
    @RequestMapping("/api")
    class FeignGreetingsClientApiGateway {

        private final GreetingsClient greetingsClient;

        @Autowired
        FeignGreetingsClientApiGateway(GreetingsClient greetingsClient) {
            this.greetingsClient = greetingsClient;
        }

        ❷
        @GetMapping("/feign/{name}")
        Map<String, String> feign(@PathVariable String name) {
            return this.greetingsClient.greet(name);
        }
    }

```

- ❶ 在 Spring profile 文件 feign 下运行边缘服务。
- ❷ 对服务的调用会成为一个方法调用，但是它仍然是依赖于 REST 的。我们的客户端



是如此地简单和直接！任何知道如何使用 Feign 的人，都可以很快了解它的实现内容。

API 适配器可以变得相当复杂、相当具体。这些适配器是一个天然的处理客户端特定逻辑（不能适用于所有请求）的地方。最终，这些为客户端适配 API 的问题，会开始变得像集成问题一样怪异。有关这方面的更多信息，请参阅第 10 章中关于集成问题的讨论。

到目前为止，我们已经了解了如何使用 REST API，来调用和处理来自其他服务的数据。为了方便理解，我们忽略了对服务之间调用可靠性的要求。有关这方面的更多信息，请参阅在第 12 章中对集成的讨论。

使用 Netflix Zuul 进行过滤和代理

到目前为止，我们讨论的主要问题是如何让指定客户端与下游 REST 服务端点集成起来。但是还有很多更常见的问题，会涉及所有从客户端访问下游服务或服务组的请求。可以使用一个微型代理 Netflix Zuul (<https://github.com/Netflix/zuul>) 来简单搭建一些过滤器，解决限速、代理等问题。



在撰写本书时，有一个名为 Spring Cloud Gateway 的 Spring Cloud 孵化器项目，该项目希望在 Spring Framework 5 的响应式运行时上，提供一个更为无缝的 Zuul 替代方案。但是因为它的代码还没有到 Generally Available (GA) 或稳定的阶段，所以我们将重点放在 Netflix 强大、通用的 Zuul 项目上，其中的原因有两点：运行良好，并且适用广泛。熟悉它会为你稍后选择使用 Spring Cloud Gateway 有帮助。

Zuul 的架构师 Mikey Cohen 在 SpringOne Platform 2016 上做了一个很好的演讲，介绍了使用 Zuul 的一些应用程序 (<http://www.slideshare.net/MikeyCohen1/zuul-netflix-springone-platform>)。Zuul 目前服务于 Netflix.com 的大部分前端和超过 1000 多种设备类型。它作为适配器层，处理着数百个不同的协议和设备版本。在 Netflix 的特定环境部署中，它前面对接着超过 50 个 AWS Elastic Load Balancer 负载均衡器，并且每天跨 3 个 AWS 区域处理数百亿的请求。在 Netflix.com 的 10 个原始系统之前，一共部署了超过 20 个 Zuul 的生产集群。总之，Zuul 是一个非常重要的组件。

Zuul 过滤器对生产环境的影响深远。它们包含动态的路由逻辑，负责处理负载均衡和服务保护，并提供调试和分析问题路径的工具。Zuul 网关可以成为一个质量保证工具，因为它可以成为一个观察数据在系统中如何流动的地方。

Zuul 过滤器可以为请求创建一个顶级上下文，处理例如地理位置、cookie 传播和令牌解密之类的问题。它们可以解决诸如认证等交叉问题，也可以用于格式化请求或响应，以



及处理某些特定设备的奇怪问题，例如分块编码、头部截断、URL 编码修正等。它们非常适合进行有针对性的路由选择，隔离特定的（尽管是错误的）请求。它们可用于处理流量整形，进行全球范围路由，处理死亡节点和区域故障切换逻辑，以及检测和防止攻击等问题。

似乎它们适用于所有类型的服务。但 Zuul 不适合处理某个特定服务的业务逻辑。不要在边缘服务中引入任何业务逻辑。

Zuul 过滤器与传统的 Servlet 过滤器（Filter）不一样。传统的 Servlet 过滤器会假定你正在过滤已经经过代理的请求，而不是发向边缘服务本身的请求。

Zuul 过滤器是指定客户端到下游服务路由的理想场所。如果你的服务已经在一个配置了 Spring Cloud DiscoveryClient 实现的服务中心注册了，那么你只需要在代码中添加一个 @EnableZuulProxy（并在类路径中添加 org.springframework.cloud:spring-cloud-starter-zuul 依赖），Spring Cloud 就会与 DiscoveryClient 一起，为你代理到下游服务的所有路由。你可以使用 RouteLocator 来查询这些路由，就像我们在示例 8-9 中做的那样。

示例8-9 ZuulConfiguration.java

```
package greetings;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
import org.springframework.cloud.netflix.zuul.filters.RouteLocator;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableZuulProxy
class ZuulConfiguration {

    @Bean
    CommandLineRunner commandLineRunner(RouteLocator routeLocator) { ❶
        Log log = LogFactory.getLog(getClass());
        return args -> routeLocator.getRoutes().forEach(
            r -> log.info(String.format("%s (%s) %s", r.getId(), r.getLocation(),
                r.getFullPath())));
    }
}
```

- ❶ RouteLocator 接口有一些有趣的实现。按照配置，Spring Cloud 会配置一个可以感知 DiscoveryClient 的实现。



Zuul 已经根据服务 ID 为我们搭建了方便的路由, 假设我们已经创建了 `service-registry` 和 `greetings-service`, 我们应该在前面例子的输出中看到, 有一个 `greetings-service` 服务可以通过代理进行消费。`greetings-client` 应该在 `http://localhost:8082` 上启动, 所以你可以调用 `http://localhost:8082/greetings-service/greet/World` 上的 `greetings-service`, 其中上下文路径 `greetings-service` 来自注册中心中的服务 ID。你也可以设置任意的路由规则。示例 8-10 显示了设置路由所需的配置。

示例8-10 `application.properties`

```
zuul.routes.hi.path = /lets/** ❶
```

```
zuul.routes.hi.serviceId = greetings-service ❷
```

❶ 边缘服务上的路由应该映射到该路径。

❷ 一个服务 (或者, 指定一个完整的 URL, 而不是使用 `serviceId` 作为 `.location` 的值)。

在该配置中, 我们将所有请求映射到 `greetings-service`, 即将路径 `/` 及以下的请求, 映射到边缘服务上的 `/lets/*` 端点。现在你可以启动应用程序, 并访问 `http://localhost:8082/lets/greet/World` 进行测试。这些路由非常适合保存在 Spring Cloud Config Server 中, 这样无须重启边缘服务就可以更新。你可以强制 Zuul 动态地重新加载它们。

实现这一点的一个方法, 就是简单地调用 Spring Cloud 的 `refresh` 端点 (我们在第 3 章中讨论过十二要素风格配置)。可以在 Spring Cloud Config Server 中更改配置, 提交更改 (Subversion 或 Git), 然后向 `refresh` 端点或 Spring Cloud 事件总线端点 (`/bus/refresh`) 发送一个空的 HTTP POST 请求 (例如, `curl -d {} http://localhost:8082/refresh`)。这将导致发布一个 `RefreshScopeRefreshedEvent` 事件, Zuul 收到该事件后会对自己进行重新配置。

或者, 你可以使用图 8-1 所示的 `/routes` Actuator 端点。Zuul 会自动为你进行配置。如果你发送了一个 HTTP GET 请求, 该端点会返回已经配置的路由; 如果你发送了一个 HTTP POST 请求, 则会触发刷新配置。这是一个代替 `refresh` 端点的方法, 但是它仅仅适用于 Zuul 路由, 而不适用于其他可能捕获刷新事件的 (更重量级的) 基础组件。



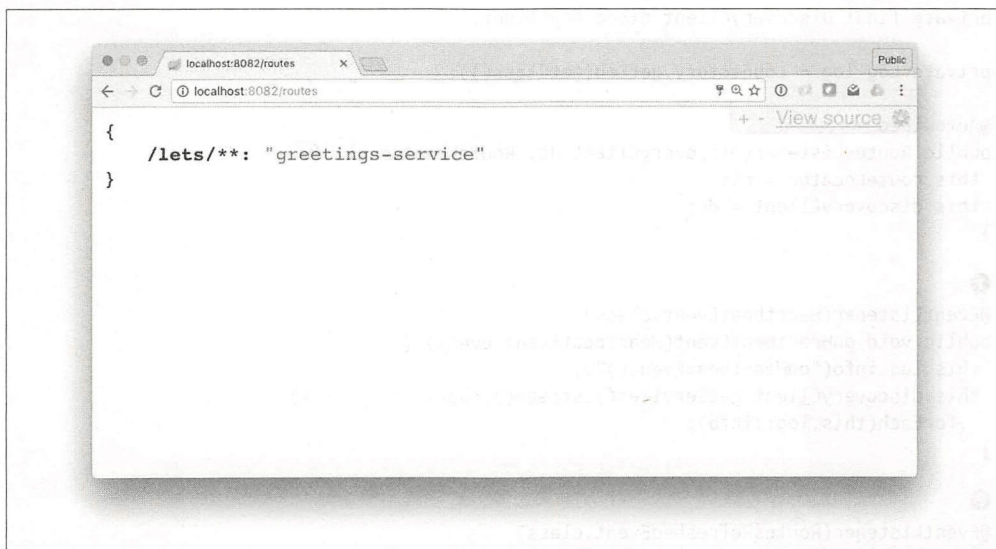


图8-1 从routes端点获得的Zuul路由

Zuul 也会根据 `DiscoveryClient` 发布的 `HeartbeatEvent` 事件，将现有路由置为失效的或者重新配置。因此，如果节点应该从 `Eureka`（或 `DiscoveryClient` 支持的任何其他服务注册中心）中注销，那么 Zuul 也会从其配置中移除该路由。

当然，如果你有某种内部状态依赖于这些路由，你可以直接监听这些事件，并以任何适当的方式进行响应（如示例 8-11 所示）。

示例 8-11 `RoutesListener.java`

```
package greetings;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.discovery.event.HeartbeatEvent;
import org.springframework.cloud.netflix.zuul.RoutesRefreshedEvent;
import org.springframework.cloud.netflix.zuul.filters.RouteLocator;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
class RoutesListener {

    private final RouteLocator routeLocator;
```



```

private final DiscoveryClient discoveryClient;

private Log log = LoggerFactory.getLog(getClass());

@Autowired
public RoutesListener(DiscoveryClient dc, RouteLocator rl) {
    this.routeLocator = rl;
    this.discoveryClient = dc;
}

❶
@EventListener(HeartbeatEvent.class)
public void onHeartbeatEvent(HeartbeatEvent event) {
    this.log.info("onHeartbeatEvent()");
    this.discoveryClient.getServices().stream().map(x -> " " + x)
        .forEach(this.log::info);
}

❷
@EventListener(RoutesRefreshedEvent.class)
public void onRoutesRefreshedEvent(RoutesRefreshedEvent event) {
    this.log.info("onRoutesRefreshedEvent()");
    this.routeLocator.getRoutes().stream().map(x -> " " + x)
        .forEach(this.log::info);
}
}

```

❶ 监听正在由 DiscoveryClient 发布的事件。

❷ 监听正在由 Zuul 路由发布的事件。

Zuul 的核心是一个代理。你可以编写 ZuulFilter 的实现，将其集成到代理自身的生命周期中，或者编写常规的 javax.servlet.Filter 实现，将请求过滤到 Zuul Servlet 中。我们来看一个简单的过滤器，它暴露了访问控制的头信息，并且允许跨源请求脚本，这样其他 JavaScript 客户端可以从其他来源调用该服务。在一个简单的应用程序中，同时提供 JavaScript 和 HTML5 资源并暴露 Zuul 代理是可以的。但是在更成熟的应用程序中，（静态的）客户端代码应该位于内容分发网络（CDN）中。这就是我们现在遇到的一个问题：从 JavaScript 代码到边缘服务（包含代理）的调用，会与防止交叉源请求伪造的保护机制相违背，因为 JavaScript 代码位于沙盒中，所以无法在请求源之外发出请求。

解决此问题的唯一方法，是在边缘服务上暴露 Access-Control-Allow-Headers、Access-Control-Allow-Methods、Access-Control-Allow-Origin 和（可选的）Access-Control-Max-Age 几个头信息，明确允许 HTML5 客户端的请求可以发往边缘服务。我们可以将来自 HTML5 客户端的请求，直接发送到在服务注册中心注册的每个服务，但是这样的话，我们需要修改每个下游微服务，一一添加这些访问控制的头信息。如果我们希望看到下游服务的效果，那就不得不要求所有相关团队更改他们的代码。这不仅可



能需要花费很长时间，同时我们也会失去对微服务的自主权。相反，我们应该为边缘服务配置一个过滤器，让它来为我们实现这一点。

最简单的办法，是在过滤器中指定 * 来允许所有的请求。不过这样会有一些限制。如果我们的 CORS 请求中包含 cookie，我们还必须在 JavaScript 代码中指定 `withCredentials:true`，并且服务必须返回 `Access-Control-Allow-Credentials:true`。但是如果这样做，我们的请求将会对所有客户端开放！另一种选择是在边缘服务上保持一个动态的白名单，过滤掉没有在服务注册中心注册的客户端。DiscoveryClient 会为我们提供所需的支持。



如果静态资源存在于 CDN 中，那么 CDN 的来源（IP 或主机名）需要出现在注册中心中。可以使用服务注册中心的 API 来实现这一点。另外，如果我们使用 Netflix Eureka，则我们可以使用 Netflix 的 Prana (<https://github.com/Netflix/Prana>) 来进行注册。

我们来看一个简单的 `javax.servlet.Filter` 示例，它会动态地将请求发送给边缘服务（如示例 8-12 所示）。

示例8-12 CorsFilter.java——你需要通过cors profile文件启动应用程序

```
package greetings;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.discovery.event.HeartbeatEvent;
import org.springframework.context.annotation.Profile;
import org.springframework.context.event.EventListener;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.http.HttpHeaders;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.net.URI;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.stream.Collectors;
```



```

@Profile("cors")
@Component
@Order(Ordered.HIGHEST_PRECEDENCE + 10)
class CorsFilter implements Filter {

    private final Log log = LoggerFactory.getLog(getClass());

    private final Map<String, List<ServiceInstance>> catalog = new ConcurrentHashMap<>();

    private final DiscoveryClient discoveryClient;

    ①
    @Autowired
    public CorsFilter(DiscoveryClient discoveryClient) {
        this.discoveryClient = discoveryClient;
        this.refreshCatalog();
    }

    ②
    @Override
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
        HttpServletResponse response = HttpServletResponse.class.cast(res);
        HttpServletRequest request = HttpServletRequest.class.cast(req);
        String originHeaderValue = originFor(request);
        boolean clientAllowed = isClientAllowed(originHeaderValue);

        if (clientAllowed) {
            response.setHeader(HttpHeaders.ACCESS_CONTROL_ALLOW_ORIGIN,
                originHeaderValue);
        }

        chain.doFilter(req, res);
    }

    ③
    private boolean isClientAllowed(String origin) {
        if (StringUtils.hasText(origin)) {
            URI originUri = URI.create(origin);
            int port = originUri.getPort();
            String match = originUri.getHost() + ':' + (port <= 0 ? 80 : port);

            this.catalog.forEach((k, v) -> {
                String collect = v
                    .stream()
                    .map(
                        si -> si.getHost() + ':' + si.getPort() + '(' + si.getServiceId() + ')'
                    )
                    .collect(Collectors.joining());
            });

            boolean svcMatch = this.catalog
                .keySet()

```




```

        .stream()
        .anyMatch(
            serviceId -> this.catalog.get(serviceId).stream()
                .map(si -> si.getHost() + ':' + si.getPort())
                .anyMatch(hp -> hp.equalsIgnoreCase(match)));
        return svcMatch;
    }
    return false;
}

④
@EventListener(HeartbeatEvent.class)
public void onHeartbeatEvent(HeartbeatEvent e) {
    this.refreshCatalog();
}

private void refreshCatalog() {
    discoveryClient.getServices().forEach(
        svc -> this.catalog.put(svc, this.discoveryClient.getInstances(svc)));
}

@Override
public void init(FilterConfig filterConfig) throws ServletException {
}

@Override
public void destroy() {
}

private String originFor(HttpServletRequest request) {
    return StringUtils.hasText(request.getHeader(HttpHeaders.ORIGIN)) ? request
        .getHeader(HttpHeaders.ORIGIN) : request.getHeader(HttpHeaders.REFERER);
}
}

```

- ① 我们将使用 Spring Cloud DiscoveryClient 来查询服务拓扑。
- ② 如你所愿，使用标准 Servlet 过滤器的 doFilter 方法。
- ③ 如果确定客户端可以请求某些服务，则设置访问控制头信息。
- ④ 主动使本地缓存失效，并在 DiscoveryClient 收到心跳事件时缓存所有已注册的服务。

完成以上步骤后，我们需要使用一个静态的 HTML5 应用程序来进行测试。这个应用程序在 html5-client 模块中，通过 Spring Cloud DiscoveryClient 来实现服务注册和发现。它暴露了一个端点 /greetings-client-uri，同时返回一个 greetings-client 边缘服务的有效 URI，这样我们的 JavaScript 代码可以向在其他节点上运行的边缘服务，发起一个 HTTP Ajax 调用（如示例 8-13 所示）。



示例8-13 Html5Client.java

```
package client;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.util.Collections;
import java.util.Map;
import java.util.Optional;

@RestController
@EnableDiscoveryClient
@SpringBootApplication
public class Html5Client {

    private final LoadBalancerClient loadBalancerClient;

    @Autowired
    Html5Client(LoadBalancerClient loadBalancerClient) {
        this.loadBalancerClient = loadBalancerClient;
    }

    public static void main(String[] args) {
        SpringApplication.run(Html5Client.class, args);
    }

    ①
    // @formatter:off
    @GetMapping(value = "/greetings-client-uri",
        produces = MediaType.APPLICATION_JSON_VALUE)
    // @formatter:on
    Map<String, String> greetingsClientURI() throws Exception {
        return Optional
            .ofNullable(this.loadBalancerClient.choose("greetings-client"))
            .map(si -> Collections.singletonMap("uri", si.getUri().toString()))
            .orElse(null);
    }
}
```

- ① 该端点会返回下游 greetings-client 的一个实例，由于边缘服务 Access-Control- * 这样的头信息，所以我们可以访问它的端点。这里，无论负载均衡使用的策略如何，我们都可以使用 LoadBalancerClient 来返回一个负载均衡器实例。默认情况下，客户端的负载均衡器会使用 Netflix Ribbon，它会使用轮询的负载均衡策略。



JavaScript 代码本身并不是什么令人兴奋的东西，它只是一个 jQuery 应用程序（还有人记得 jQuery 吗？）使用从 `html5-client` 应用程序中解析出的 URI 来调用边缘服务（`greetings-client`），而边缘服务会相应调用 `greetings-service` 服务。应用程序需要更新本地页面的 DOM 才能看到响应信息。



HTML5 应用程序通过 WebJars 项目 (<http://www.webjars.org>) 来引入 jQuery。WebJars 可以让我们在 Maven 构建中管理 JavaScript 依赖关系。为了引入 jQuery，只需要添加两个依赖到构建中：`org.webjars:jquery:2.1.1`，它引入了 jQuery 库本身，以及 `org.webjars:webjars-locator`，它可以让 Spring Boot 和其他基于 JVM 的 Web 框架，将版本不明的 JavaScript 库（`<script src ="/ webjars / jquery / jquery.min.js"> </ script>`）对应到类路径上具体版本的库。

示例 8-14 显示了代码。

示例8-14 使用jQuery的代码

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8"/>
  <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
  <title>Demo</title>
  <meta name="description" content=""/>
  <meta name="viewport" content="width=device-width"/>
  <base href="/"/>
</head>
<body>

<div id="message"></div>

<script src="/webjars/jquery/jquery.min.js"></script>
<script>

  ❶
  var greetingsClientUrl = location.protocol + "/" + window.location.host
    + "/greetings-client-uri";
  $.ajax({url: greetingsClientUrl}).done(function (data) {
    var nameToGreet = window.prompt("who would you like to greet?");
    var greetingsServiceUrl = data['uri'] + "/lets/greet/" + nameToGreet;
    console.log('greetingsServiceUrl: ' + greetingsServiceUrl);
    ❷
    $.ajax({url: greetingsServiceUrl}).done(function (greeting) {
      $("#message").html(greeting['greeting']);
    });
  });
</script>
</body>
</html>
```

❶ 加载当前主机信息，找到 `greetings-client`。

❷ 使用已解析的 URI 调用跨源服务。

要查看所有功能是否正常，请按照以下顺序启动（必须按此顺序，否则，你不得不等待所有服务都注册到服务注册中心上）：`service-registry`、`greetings-service`、`edge-service`（使用 `cors profile`）以及 `html5-client`。打开 Eureka（<http://localhost:8761>），找到 HTML5 客户端实例现在可用的 IP，然后将 IP 粘贴到浏览器中，添加端口 8083，如图 8-2 所示。



使用正确的主机和端口访问 HTML 5 客户端很重要；否则，`CorsFilter` 不会允许请求通过，因为它会检查所有向服务注册中心注册的主机和端口发送的请求。（例如，这可能无法适用于 `localhost` 访问。）

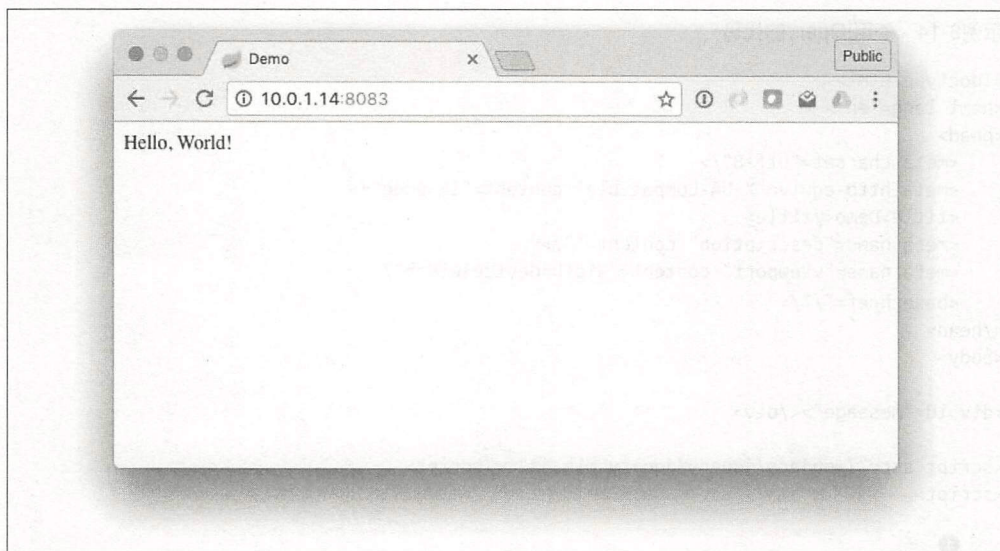


图8-2 来自HTML5客户端的CORS请求

目前还不错！一个简单的 `Servlet Filter` 可以为我们完成很多很酷的事情。但是，`Servlet API` 无法知道我们正在调用哪个代理的下游服务。如果我们希望它能够意识到，需要编写一个 `Zuul` 过滤器。

自定义 Zuul 过滤器

`CorsFilter` 是一个标准的过滤器，它适用于所有针对边缘服务的请求，包括所有发给

Zuul（以及 Spring Boot 自动创建的 Zuul Servlet）的请求。也就是说，Zuul 有一个专门的流水线，用来过滤所有通过 Zuul 代理路由的请求。Zuul 过滤器默认有四种类型，当然如果需要的话，你也可以添加自定义的类型：

- *pre* 过滤器在请求路由之前执行。
- *routing* 过滤器可以处理请求的实际路由。
- *post* 过滤器在请求路由之后执行。
- 如果在处理请求的过程中发生错误，则会执行 *error* 过滤器。

假定满足了相关的自动配置条件，Spring Cloud 会自动注册几个有用的 Zuul 过滤器。

- `AuthenticationHeaderFilter` (**pre**)：用于查找正在进行代理的请求，并在向下游发送之前删除 `authorization` 标头。
- `OAuth2TokenRelayFilter` (**pre**)：用于传播请求中可用的 OAuth 访问令牌。
- `ServletDetectionFilter` (**pre**)：检测某个 HTTP servlet 请求是否已经通过了 Zuul 过滤器管道。
- `Servlet30WrapperFilter` (**pre**)：用 Servlet 3.0 装饰器包装一个 HTTP 请求。
- `FormBodyWrapperFilter` (**pre**)：用含有关于 multipart 文件上传元数据的装饰器，包装一个 HTTP 请求。
- `DebugFilter` (**pre**)：如果请求包含一个 `Archaius` 属性，则提供参数支持调试。
- `SendResponseFilter` (**post**)：提供一个输出结果的响应（如果上游过滤器已经提供了一个输出结果）。
- `SendErrorFilter` (**post**)：如果回复中包含一个错误，则将其转发给一个可配置的端点。
- `SendForwardFilter` (**post**)：如果回复包含一个转发，则进行必要的转发。
- `SimpleHostRoutingFilter` (**route**)：接收传入的请求，并根据 URL 决定将代理请求路由到哪个节点。
- `RibbonRoutingFilter` (**route**)：接收传入的请求，并根据 URL 决定将代理请求路由到哪个节点。它使用 Netflix Ribbon 的可配置路由和客户端负载均衡。

假设我们想要增加另一个常见的横切面关注点，限速器可能是一个较简单的例子。其可以限制向下游服务发送的请求数，从而帮助系统确保 SLA。令牌桶算法 (https://en.wikipedia.org/wiki/Token_bucket) 基于将令牌按照一个固定速度，放到一个固定容量的桶中，其中令牌通常代表一个字节单位或一个固定大小的包。另一种算法，漏桶算法 (https://en.wikipedia.org/wiki/Leaky_bucket) 是模拟一个带有漏口的桶，如果进水的速度超过水漏出的速度，或者一次进入超过桶容量的水，那么桶中的水都会溢出来。

简单的限速器很容易通过 Zuul 过滤器实现。我们这个实现使用了 Guava `RateLimiter` 类。

但是它不太实用，因为它限制的是所有代理服务和服务实例之间的请求，而不是到每个服务实例甚至指定服务的请求。要运行它，我们必须首先在示例 8-15 中配置 Guava RateLimiter。

示例8-15 Guava RateLimiter被配置为每10s只允许一次请求；这是一个不合理的低值，仅仅用来演示其中的过程

```
package greetings;

import com.google.common.util.concurrent.RateLimiter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Profile("throttled")
@Configuration
class ThrottlingConfiguration {

    @Bean ❶
    RateLimiter rateLimiter() {
        return RateLimiter.create(1.0D / 10.0D);
    }
}
```

- ❶ 指定每秒允许的请求数量。在这里，我们指定每秒允许 0.10 个请求。或换句话说，每 10 秒允许一个请求。

有了这个配置之后，我们可以轻松设计一个限制传入请求数量的 ZuulFilter（如示例 8-16 所示）。

示例8-16 一个 rate-limiting ZuulFilter

```
package greetings;

import com.google.common.util.concurrent.RateLimiter;
import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.core.Ordered;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.util.ReflectionUtils;

import javax.servlet.http.HttpServletResponse;
```



```

@Profile("throttled")
@Component
class ThrottlingZuulFilter extends ZuulFilter {

    private final HttpStatus tooManyRequests = HttpStatus.TOO_MANY_REQUESTS;

    private final RateLimiter rateLimiter;

    @Autowired
    public ThrottlingZuulFilter(RateLimiter rateLimiter) {
        this.rateLimiter = rateLimiter;
    }

    ❶
    @Override
    public String filterType() {
        return "pre";
    }

    ❷
    @Override
    public int filterOrder() {
        return Ordered.HIGHEST_PRECEDENCE;
    }

    ❸
    @Override
    public boolean shouldFilter() {
        return true;
    }

    ❹
    @Override
    public Object run() {
        try {
            RequestContext currentContext = RequestContext.getCurrentContext();
            HttpServletResponse response = currentContext.getResponse();

            if (!rateLimiter.tryAcquire()) {

                ❺
                response.setContentType(MediaType.TEXT_PLAIN_VALUE);
                response.setStatus(this.tooManyRequests.value());
                response.getWriter().append(this.tooManyRequests.getReasonPhrase());

                ❻
                currentContext.setSendZuulResponse(false);

                throw new ZuulException(this.tooManyRequests.getReasonPhrase(),
                    this.tooManyRequests.value(), this.tooManyRequests.getReasonPhrase());
            }
        }
    }
}

```

```

catch (Exception e) {
    ReflectionUtils.rethrowRuntimeException(e);
}
return null;
}
}

```

- ❶ 这个过滤器是一个在代理请求之前运行的 pre 过滤器。
- ❷ 过滤器应尽早运行。
- ❸ 它应该始终运行。不过，你可能会根据请求的某个属性或配置维度，禁用过滤器。
- ❹ run() 方法是 ZuulFilter 的核心。在这里，你需要完成过滤某个指定请求的工作。
- ❺ 过滤器会尝试从 Guava RateLimiter 获取一个许可证，如果失败，它会返回一个 HTTP 状态码 429 和一个错误消息，表示请求过多。
- ❻ 显式取消代理请求。如果不这样做，请求会被自动代理。

通过 profile 文件 throttled 来运行 greetings-client 边缘服务。如果你这时访问 greetings-client 服务，比如 `http://localhost:8082/lets/greet/Eva`，那么你将每 10 秒钟收到一次有效的 JSON 响应，否则你会得到一个表示太多请求的响应。

边缘服务的安全

边缘服务代表防御外部恶意请求的第一道防线。边缘服务是处理交叉问题（例如安全）的有效位置。安全问题是令人乏味的，因为每个服务都需要处理安全问题，所以最后可能会重复处理。安全性会让一个平台（例如 Cloud Foundry）变得更强大，而 Spring Boot 约定大于配置的方式对于安全很有帮助。当请求进入系统时，我们会在边缘服务进行身份认证，然后将身份认证上下文传播给下游服务。

身份认证仅仅意味着我们要确定发起指定请求的角色。是使用用户名和密码的方式？还是 x509 证书？如果只是试图回答“谁发出了这个请求？”的问题，那么这些方案及类似方案似乎是足够的了，当然，要正确使用。但是，正确使用它们是很难的（<http://bit.ly/2vnx3DD>）。理想情况下，你应该对哈希进行调优，以便花费最少的时间来进行认证（也许 0.5s）。这意味着认证用户名和密码的最少时间是 0.5s（加上所有的实际工作）。密码很难控制谁可以访问系统，因为如果密码泄漏了，那么所有的密码使用都是无效的。人们也可以跨系统共享密码。通常，谁是发起请求的人并不是你要回答的唯一问题，即使你已经确保密码足够随机并且已经安全地保存起来（实际可能并不如此）。

在如今的情况中，并不是所有的客户端都是平等的。了解用户发起请求使用的客户端与知道谁发起请求几乎同等重要。当我们为不同客户端提供服务时，需要能够同时知道这两个方面。API 提供商可能会更信任某些客户端，并相应限制其他客户端的访问。

举一个例子，想想你所有使用 Facebook 的场景。Facebook 提供了众多的客户端：iPad 端、网页端，以及 Android 应用程序等。你能够经常在许多网站上发现“用 Facebook 登录”的按钮选项。单击这个按钮会将跳转到 Facebook.com 上，如果你尚未登录，网站会提示你登录，然后它会提示你为请求的网站批准某些权限。确认访问权限之后，Facebook 会将你重定向回原始的网站，这时该网站已经拥有访问你的 Facebook 个人资料信息所需的权限，并会自动填写你的个人信息，快速进行登录。在本例中，你完全不需要在第三方网站中输入你的 Facebook 密码。第三方网站只有有限的操作权限，只能获取某些信息，或者做某些事情。如果你决定要收回该第三方网站访问个人资料的权限，只需要登录 Facebook 并撤销对该网站的授权即可。重要的是，你不需要更改你的密码。密码仍然有效，并且所有其他已连接的客户端仍然有效。这里的区别在于，你的账户（但仅限于通过第三方网站（客户端）访问）不再有效。

我们将其与使用 Facebook 官方 iPhone 应用程序的体验相比较。iOS 应用程序是由 Facebook 的工程师开发的。当然，你不必担心他们会滥用你的信任，使用你的 Facebook.com 上的个人资料信息和账户做一些不好的事情。在这种情况下，Facebook.app 在你自己的 iOS 设备上运行，因此更偏向让你在设备上输入用户名和密码。它不需要在浏览器中将你重定向到 Facebook.com，也不应该这样做。Facebook.com 已经拥有你的密码，他们也不希望信息被 iPhone 应用的登录屏幕泄露出去。

这就是所有的内容，两个不同的 Facebook 客户端（其中一个更具有信任度）以及一个个人档案。每个客户端有不同的权限。官方的 Facebook iPhone 应用程序基本上可以完全控制你的账户，而第三方网站可能只能读取你的电子邮件和全名。

OAuth

现在我们进入 OAuth 的话题。OAuth 是一个标准，有三个版本：1.0、1.0.a 和 2.0。OAuth 2.0 是最新、最推荐的版本，所以我们主要关注 OAuth 2.0。OAuth（简称“开放授权”）是互联网上基于令牌的授权标准。令牌减少了用户名和密码暴露的时间窗口。令牌可以将客户端与密码分离，确保错误的客户端永远无法锁定你的账户。令牌可以表示某个用户使用的客户端，或者表示一个没有某个用户的上下文。

OAuth 是一个授权协议（“用户具有什么权限？”），而不是一个身份认证协议（“这个用户是谁？”），所以你仍然需要在某个地方处理身份认证，然后将控制权转交给 OAuth。

OAuth 为客户端提供了一个访问服务的“安全委托访问”，作为服务的数据所有者。通过身份认证后，客户端可以使用令牌来授权请求。令牌是与客户端相关的。

OAuth 中的一些术语。

- **客户端**：一个发起保护请求的应用程序。客户端经常（但并非总是）代表最终用户（资源所有者）发起请求。它可能是一个 iPhone 或 Android 应用程序、HTML5 浏览器客户端，甚至是智能手表！不过，客户端不必一定拥有用户上下文。例如，客户端可能是一个处理后台的分析工作的程序，需要有限的访问权限。在 Facebook.com 示例中，它可能是你手机上的 iOS 应用程序或者桌面 Web 浏览器。
- **资源所有者**：通常指允许客户端访问他们的信息的最终用户（“Juergen”、“Michelle”、“Dave”、“Margarette”等）。在 Facebook.com 的例子中，可能是你或我。
- **资源服务器**：托管受保护资源的服务器，能够接收和响应包含访问令牌的受保护请求。这指的是客户端调用的受保护的 API。在 Facebook.com 的例子中，它指的是支持访问所有信息（你的个人资料、社交图谱等）的 Facebook API。
- **授权服务器**：服务器在成功认证资源所有者并获得授权后，会向客户端颁发访问令牌。在 Facebook.com 中，是你被重定向后进行身份认证并提供令牌的服务。对于 Facebook.com 来说，授权服务器同时也是资源服务器，但是它们并不必须是同一个。

虽然在 OAuth 中第一步是获得授权，但是基本上要首先通过身份认证。有 4 个众所周知的授权类型或流程，能够实现这一点。

服务端应用程序

在上面描述的“使用 Facebook 登录”流程中，客户端从第三方网站被重定向到 Facebook.com 的登录界面。在浏览器中，你会看到地址栏中的安全锁图标，表示当前网站通过了 HTTPS 认证。你可以放心并确认访问的是 Facebook.com。然后输入你的用户名和密码并提交表单，系统会提示你为请求客户端授予某些权限。如果你单击 Allow 按钮，Facebook.com 会通过一个授权码重定向到发起请求的客户端。客户端的服务端会调用 Facebook.com 的授权服务器，同时提交授权码来交换访问令牌。这样，发起请求的客户端现在就有了一个访问令牌，它会将访问令牌存储在数据库中，并使用它来调用 Facebook.com 的资源服务器。

在这种情况下，访问令牌永远不会被泄漏给浏览器中的 JavaScript 代码。客户端（或正在使用客户端的用户）唯一可见的是授权码。这种间接的方式可以防止中间人攻击，在这种攻击中，有人可以拦截到访问令牌，并使用该令牌来代表你进行调用。

这个流程被称为授权码模式。如果你想防止访问令牌被泄漏出去，这种模式很有用。

HTML5 和 JavaScript 单页面应用程序

从网页加载源代码后，单页面应用程序（SPA）会完全在浏览器中运行。当然，SPA 不能用来保存任何私密信息，因为通过单击 View Source 可以看到它的源代码。在这个流程中，用户单击 Sign In 按钮，提示批准其访问，就像以前一样。如果用户单击 Allow 按钮，则服务会将用户重定向回单页应用程序站点，并在 URL 片段中附带令牌（URL 中 # 之后的部分，例如 `http://some-third-party-service.com/an_oauth_callback#token=12345 ..`）。JavaScript 应用程序能够读取这部分 URL，并且更改 URL 地址不会让浏览器发出新的请求。这被称为简化模式。在这种情况下，由于没有服务端组件，因此该流程只会将令牌泄露给客户端代码。

没有用户的应用

到目前为止我们看到的所有例子，我们介绍的客户端，都是一个用户来访问该用户受保护的资源。对于客户端来说，即使没有任何用户，也可能仅通过客户端凭证来获取一个访问令牌。这对于那些需要访问受保护信息，但是又没有特定用户上下文的应用程序，例如批处理、分析程序或者任何非交互式的应用程序都很有用。这类客户端可以通过认证获得有限的权限。这被称为客户端模式。

受信任的客户端

对于可信任的客户端，OAuth 2 支持密码授权的方式。试想一下，你正在为 Facebook.com 开发 Facebook.app。如果用户需要在 Facebook.app 内部重定向到 Facebook.com，然后批准 Facebook.app 有权读取和操作你的 Facebook.com 上的数据，这样的流程似乎有点多余（因为 Facebook.com 已经拥有你的用户名和密码）。在这种情况下，由 Facebook 工程师开发的客户端是可被信任的，所以只需传输用户的用户名和密码即可。Facebook.app 不会突然要求获得你的用户名和密码，当然，它们已经有了！事实上，如果你怀疑 Facebook.app 会对你的 Facebook.com 数据有恶意行为，那么你可能需要重新考虑是不是应该使用 Facebook！在这个流程中，用户会直接向授权服务器发送用户名和密码来换取访问令牌。这被称为密码模式授权。

继续更深入讨论 OAuth 已经超出了本书的范围。这里推荐有关 OAuth 的一本优秀（而且简明）书籍，Ryan Boyd 编写的 *Getting Started With OAuth 2.0*（O'Reilly 出版）。

我们将建立一个 OAuth 授权服务器，并使用 OAuth 来保护我们的客户端，但首先，我

们来回顾一下 Spring Security 的基础知识。

Spring Security

我们曾说过,OAuth 是一个身份认证委托协议,它需要回答一个问题:“他(她)是他(她)自己吗?”我们需要使用 Spring Security 来回答这个问题。Spring Security 几乎可以与所有的身份提供方集成在一起,即使没有,自己编写集成代码也非常容易。

身份认证主要与“谁”发起请求有关,是 dsyer 还是 jhoeller? 在 Spring Security 中,一般由 AuthenticationManager 接口的实现来处理认证过程(如示例 8-17 所示)。

示例8-17 org.springframework.security.authentication.AuthenticationManager

```
public interface AuthenticationManager {  
  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
}
```

如果身份认证的请求是有效的,AuthenticationManager 实例会返回一个 Authentication 实例,其 authenticated 属性被设置为 true。如果请求无效,则抛出一个 AuthenticationException 异常,如果无法确定,则返回 null。在这个层次上,AuthenticationManager 不会关心身份提供方,也不会关心请求本身,无论它们是 HTTP 请求、本地方法调用、RPC 服务调用还是异步消息等,更不会关心进行身份认证的方式,无论请求中是否包含令牌、用户名和密码以及证书等。AuthenticationManager 故意是不准确的。

ProviderManager 是 AuthenticationManager 的一个通用实现,它会委托给一系列 AuthenticationProvider 的实现。AuthenticationProvider 实现的工作方式与 AuthenticationManager 几乎完全相同,只是它们还可以处理指定的 Authentication 的 Class<?>。这使得我们可以更有效地、选择性地处理各种身份认证请求。ProviderManager 意味着 AuthenticationManager 可以通过委托的手段来处理不同的身份认证类型。

ProviderManager 对象本身有一个实现了 AuthenticationManager 的父类,如果没有任何 AuthenticationProvider 实现能够认证请求,则由该父类的 AuthenticationManager 实现来处理。我们可以使用一个全局的 AuthenticationManager,然后嵌套一些 ProviderManager 实例来专门保护某些资源。有一些 ProviderManager 实例可以对接像 LDAP 这类服务或者一个后端的 javax.sql.DataSource 数据库,用于身份认证过程中的信息查询。对于后一种情况,即从数据库中查询指定用户信息,Spring Security 已经提

供了针对性的 `AuthenticationProvider` 实现 `DaoAuthenticationProvider`，后者又会委托给一个 `UserDetailsService` 接口的实现（如示例 8-18 所示）。

示例8-18 `org.springframework.security.core.userdetails.UserDetailsService`

```
public interface UserDetailsService {  
  
    UserDetails loadUserByUsername(String username)  
        throws UsernameNotFoundException; ❶  
}
```

❶ 返回一个 `UserDetails` 实现或抛出异常。永远不应该返回 `null`。

这里的约定很简单：返回指定用户名的 `UserDetails` 对象。`UserDetails` 应该封装一些 `Spring Security` 能够通过用户名和密码认证用户身份的信息。用户的用户名是什么？用户的密码是什么？该用户的账户是否仍处于激活状态，未被锁定或者已过期？用户有什么权限？尤其是一个 `GrantedAuthority` 类型的 `authority` 字段，其内容实际上是一个字符串，它的含义会随着不同系统发生变化。它可能表示系统的授权定义、范围、权限、角色或者其他概念。但是，一旦你使用 `GrantedAuthority`，就意味着在进行授权，而不是身份认证。

授权是关于确定一个请求者的权限，即该用户允许和不允许执行的操作。这里的核心接口是 `AccessDecisionManager`。对于由一个身份认证对象、一个配置属性（`ConfigAttribute` 实例）集合，以及传入的 `object` 上下文所组成的集合，`AccessDecisionManager` 会对该集合做出一个访问控制决定。`ConfigAttribute` 集合中包含了请求的通用参数，甚至可能是使用 `Spring` 表达式语言（`Spring Expression Language`）的语句。`object` 参数是一个做出访问决定所需要的上下文。它可以是用户想要访问的任何东西，包括一个方法调用、一个受保护的 `HTTP` 端点或者一个 `Message <T>` 对象等。配置属性和上下文组合在一起，`AccessDecisionManager` 就可以决定是否授权（如示例 8-19 所示）。

示例8-19 `org.springframework.security.access.AccessDecisionManager`

```
public interface AccessDecisionManager {  
  
    void decide(Authentication authentication, Object context,  
        Collection<ConfigAttribute> configAttributes)  
        throws AccessDeniedException, InsufficientAuthenticationException; ❶  
  
    boolean supports(ConfigAttribute attribute);  
    boolean supports(Class<?> clazz);  
}
```

❶ 如果请求无法被授权，则抛出异常。

AccessDecisionManager 反过来会委托给 DecisionVoter 实例，就如同 AuthenticationManager 会委托给 AuthenticationProvider 实例一样（如示例 8-20 所示）。

示例8-20 org.springframework.security.access.AccessDecisionVoter

```
public interface AccessDecisionVoter<S> {  
  
    int ACCESS_GRANTED = 1;  
    int ACCESS_ABSTAIN = 0;  
    int ACCESS_DENIED = -1;  
  
    boolean supports(ConfigAttribute attribute);  
  
    boolean supports(Class<?> clazz);  
  
    int vote(Authentication authentication,  
            S object,  
            Collection<ConfigAttribute> attributes);  
}
```

好吧！你可能要重读几次上面介绍 Spring Security 的内容。如果你没有完全理解，也没有关系！你真正需要从这些内容中理解的是，Spring Security 使用两个不同的但是逻辑上一致的类型层级来处理认证和授权。在这两个层级结构中，都有一个根接口，它的实现会将真正的处理逻辑委托给多个链式的、类似于根接口的实例。这意味着双方都支持责任链模式。

所有这些原理都是非常底层的。在 Spring Boot 的应用程序中，你不需要知道，更不用说安装和配置大多数这些类型。即使没有 Spring Boot，底层的 Spring Security 也已经为大部分类型设置了默认值。无论如何，在保护 Web 应用程序安全时，理解底层原理是有用的。在 Web 层中，Spring Security 会以一个单独的 javax.servlet.Filter 存在（从 Web 容器的角度来看），然后委托给 Filter 实例的其他虚拟链（只有 Spring Security 知道）。每个受保护的 Web 端点都可以拥有自己的过滤器链。这些虚拟过滤器的顺序很重要，特殊规则应该放在通用过滤器之前。例如，响应 /api/* * 请求的链应该出现在 /** 的链之前。稍后，当我们讲述如何保护一部分边缘服务时，将了解如何限制对部分 Web 应用程序的访问，并指出如何处理未经身份认证的请求，以及如何让访问受保护资源的请求登录和注销。

在了解了这些背后的知识以后，我们来重新回顾 OAuth 实现。需要回答这个问题，谁在发起请求？我们可以配置和插入任意的 AuthenticationManager 实现，但出于考虑，我们假设有一个含有用户名和密码的数据库。我们可以插入一个 UserDetailsService 的实现，它通过 JPA（或其他任何技术）来认证用户名和密码。我们将创建一个 JPA 实体 Account，我们希望在接收到身份认证请求后，去查询相应的 Account 对象（如示例 8-21 所示）。



和在其他一些章节一样,我们将使用编译时注解处理器 Project Lombok (<https://projectlombok.org>), 来简化代码中成员变量的访问函数、构造函数等。

示例8-21 Account

```
package auth.accounts;
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Account {
```

```
    @Id
    @GeneratedValue
    private Long id;
```

```
    private String username, password; ❶
```

```
    private boolean active; ❷
```

```
    public Account(String username, String password, boolean active) {
        this.username = username;
        this.password = password;
        this.active = active;
    }
}
```

- ❶ 需要提供的用户名和密码。
- ❷ 当实现 UserDetailsService 时, 需要回答四次同样的问题, 这个账户是处于激活状态吗?

我们需要一个 Spring Data repository, 来简化与数据库中 Account 记录的交互过程 (如示例 8-22 所示)。

示例8-22 AccountRepository

```
package auth.accounts;

import org.springframework.data.jpa.repository.JpaRepository;
import java.util.Optional;

public interface AccountRepository extends JpaRepository<Account, Long> {

    ❶ Optional<Account> findByUsername(String username);
}
```

- ❶ 当实现 UserDetailsService 实例时，需要根据指定的 username 进行查询，所以我们让 Spring Data repository 来解决这个问题。

最后，提供一个 UserDetailsService 实现。如果在 Spring 应用程序上下文中检测到实现类，Spring Security 会自动注入它。这个实现会将我们 repository 中的 Optional<Account>，映射到 UserDetails（在 Spring Security 中称为一个 User）的具体实现上，并分配一些静态的 GrantedAuthority 实例。



在更复杂的示例中，可以根据指定的条件，或者从数据库中读取条件记录，来动态地设置 GrantedAuthority 实例。

示例 8-23 提供了一个可以感知 Account 类的 UserDetailsService 实例。

示例8-23 AccountConfiguration.java

```
package auth.accounts;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;

@Configuration
public class AccountConfiguration {

    @Bean
    UserDetailsServiceImpl userDetailsService(AccountRepository accountRepository) {
        ❶ return username -> accountRepository
            .findByUsername(username)
    }
}
```



```

.map(
    account -> {
        boolean active = account.isActive();
        return new User(account.getUsername(), account.getPassword(), active,
            active, active, active, AuthorityUtils.createAuthorityList("ROLE_ADMIN",
                "ROLE_USER"));
    })
.orElseThrow(
    () -> new UsernameNotFoundException(String.format("username %s not found!",
        username)));
}
}

```

- ① UserDetailsServiceImpl 实现的约定很简单：给定一个字符串 username，返回一个 UserDetails 实现，或者抛出一个 UsernameNotFoundException 异常。但是，在任何情况下，它都不应该返回 null。

完成这些工作之后，我们只是完成了与 Spring Security 的集成配置。我们的目标是通过这个配置来认证来自用户的令牌，但是这个配置也支持使用 HTTP BASIC 或者 HTML 登录表单的方式进行身份认证。这是 Spring Security 的基础，即使在十多年前还没有云原生的时代，你也会写出类似的代码。

Spring Cloud Security

Spring Cloud Security 让我们很容易在微服务中集成安全功能。它可以与任何符合 OAuth 协议的授权服务器通信，以及声明式地保护资源服务器。虽然我们只想保护自己的 REST API，但首先我们使用 Spring Security OAuth 来搭建一个 OAuth 授权服务器。

一个 Spring Security OAuth 授权服务器

我们将建立一个新的微服务，它会通过一个 auth-service 模块来处理授权工作。这个模块需要在类路径中引入 org.springframework.cloud:spring-cloud-starter-config、org.springframework.cloud:spring-cloud-starter-eureka、org.springframework.cloud:spring-cloud-starter-oauth2、org.springframework.boot:spring-boot-starter-data-jpa、org.springframework.boot:spring-boot-starter-web、org.springframework.boot:spring-boot-starter-actuator 这几个依赖。授权服务器会与服务注册中心（我们的 Netflix Eureka 实例）配合使用，因此还需要使用 @EnableDiscoveryClient 注解来完成服务注册和发现。



像 Spring Cloud Security 授权服务器这样的技术，在架构中起着重要的作用。在本节中，我们将试着创建一个简化的版本，但重要的是理解这些功能与业务无关。理想情况下，这应该由第三方提供。像 Okta 或 Cloud Foundry 这种支持 OAuth 的第三方身份提供商，可以提供基于 Spring Security Authorization Server 的 User Account Authentication (UAA) Server。实现安全性很难，尽可能选择专业的解决方案！

OAuth 需要知道如何认证用户，于是我们集成了 `UserDetailsService`，它需要了解可能连接它的客户端，以及这些客户端拥有的权限。我们必须先描述清楚这些客户端。正如我们使用 `Account` 实体为用户建模一样，我们将一个 `Client` 实体作为 Spring Security OAuth 的客户端（如示例 8-24 所示）。

示例 8-24 Client.java

```
package auth.clients;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.util.StringUtils;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity

public class Client {

    @Id
    @GeneratedValue
    private Long id;

    private String clientId;

    private String secret;

    private String scopes = StringUtils
        .arrayToCommaDelimitedString(new String[] { "openid" });

    private String authorizedGrantTypes = StringUtils
        .arrayToCommaDelimitedString(new String[] { "authorization_code",
            "refresh_token", "password" });

    private String authorities = StringUtils
        .arrayToCommaDelimitedString(new String[] { "ROLE_USER", "ROLE_ADMIN" });
```



```
private String autoApproveScopes = StringUtils
    .arrayToCommaDelimitedString(new String[] { ".*" });

public Client(String clientId, String clientSecret) {
    this.clientId = clientId;
    this.secret = clientSecret;
}
}
```

稍后，我们需要通过它的 `clientId` 找到一个客户端，所以我们需要一个 Spring Data JPA repository 以及一个自定义的查找方法（如示例 8-25 所示）。

示例8-25 ClientRepository.java

```
package auth.clients;

import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface ClientRepository extends JpaRepository<Client, Long> {

    Optional<Client> findByClientId(String clientId); ❶
}
```

❶ 通过它的 `clientId` 找到一个客户端。

许多流行的服务，都提供了注册临时客户端的能力，注册后即允许客户端使用服务的 API。例如，Facebook 和 Twitter 允许在开发者门户上注册第三方客户端。你可以使用刚刚创建的 Spring Data repository，轻松创建你自己的客户端。事实上，如果你给 Spring Data repository 加上注解，允许它使用 Spring Data REST，你甚至可以允许程序化的、基于 REST 的客户端注册。无论如何，这比暴露一个自己的服务，供公司所有团队注册自己的客户端和权限更简单。对于我们来说，将客户端实现细节硬编码到代码中已经足够了。

Spring Security OAuth 也需要进行一些配置。它需要知道如何认证用户，所以我们必须将它指向一个有效的 `AuthenticationManager` 实例（当我们安装自定义的 `UserDetailsService` 时，就会自动建立一个 `AuthenticationManager` 实例，使用这个实例就可以了），并且我们必须将它指向 `ClientDetailsService`，这些我们都会在 `AuthorizationServerConfiguration` 中完成。

我们必须修改 `Client` 实例来实现 `ClientDetails` 接口，就像我们让 `Account` 实体实现 `UserDetailsService` 接口一样。在示例 8-26 中，我们将使用一个特定的 `ClientDetails` 类——`BaseClientDetails` 来完成这项工作。

示例8-26 ClientConfiguration.java

```
package auth.clients;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.client.discovery.DiscoveryClient;
import org.springframework.cloud.client.discovery.event.HeartbeatEvent;
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.event.EventListener;
import org.springframework.security.oauth2.provider.ClientDetailsService;
import org.springframework.security.oauth2.provider.ClientRegistrationException;
import org.springframework.security.oauth2.provider.client.BaseClientDetails;

import java.util.Collections;
import java.util.List;
import java.util.Optional;
import java.util.Set;
import java.util.concurrent.ConcurrentSkipListSet;
import java.util.stream.Collectors;

@Configuration
public class ClientConfiguration {

    private final LoadBalancerClient loadBalancerClient;

    @Autowired
    public ClientConfiguration(LoadBalancerClient client) {
        this.loadBalancerClient = client;
    }

    @Bean
    ClientDetailsService clientDetailsService(ClientRepository clientRepository) {
        return clientId -> clientRepository
            .findByClientId(clientId)
            .map(
                client -> {

                    BaseClientDetails details = new BaseClientDetails(client.getClientId(),
                        null, client.getScopes(), client.getAuthorizedGrantTypes(), client
                            .getAuthorities());
                    details.setClientSecret(client.getSecret());

                    ❶
                    // details.setAutoApproveScopes
                    // (Arrays.asList(client.getAutoApproveScopes().split(",")));

                    ❷
                    String greetingsClientRedirectUri = Optional
                        .ofNullable(this.loadBalancerClient.choose("greetings-client"))
                        .map(si -> "http://" + si.getHost() + ':' + si.getPort() + '/')
                }
            );
    }
}
```



```

        .orElseThrow(
            () -> new ClientRegistrationException(
                "couldn't find and bind a greetings-client IP"));

        details.setRegisteredRedirectUri(Collections
            .singleton(greetingsClientRedirectUri));
        return details;
    })
    .orElseThrow(
        () -> new ClientRegistrationException(String.format(
            "no client %s registered", clientId)));
    }
}

```

- ❶ 考虑 Facebook.com 的例子。当客户端请求一个令牌时，会提示它们登录并明确地授予所有已请求的作用域。如果我们将 `autoApproveScopes` 设置为正在请求的作用域名称，那么只需要通过身份认证就可以授予所有已请求的作用域。
- ❷ 客户端应该指定一个 URI，当用户同意请求作用域后，授权服务应该重定向到该 URI。在这种特殊情况下，因为所有客户端都使用相同的服务注册中心，所以我们可以通过服务注册和 `LoadBalancerClient`，自动发现负载均衡之后的 URI。在这个例子中，尽管我们可以把 URI 保存在客户端的 JPA 记录中，但是我们已经根据客户端的需要对 URI 进行了硬编码。

`AuthorizationServerConfiguration` 继承自 `AuthorizationServerConfigurerAdapter`，而后者又实现了 `AuthorizationServerConfigurer`。`AuthorizationServerConfigurer` 是 Spring Security OAuth 在初始化生命周期的适当阶段调用的回调接口，这使得我们有机会对服务行为的各个部分进行配置。我们可以重写 `configure(ClientDetailsServiceConfigurer clients)` 方法来配置客户端，并覆盖 `configure(AuthorizationServerEndpointsConfigurer endpoints)` 方法，将 Spring Security OAuth 和之前配置的 `AuthenticationManager` 连接起来。这就是 Spring Security OAuth 将身份认证委托给 Spring Security（如示例 8-27 所示）的地方。

示例8-27 `AuthorizationServerConfiguration.java`

```

package auth;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;

//@formatter:off
import org.springframework.security.oauth2
    .config.annotation.configurers.ClientDetailsServiceConfigurer;
import org.springframework.security.oauth2

```

```

        .config.annotation.web.configuration.AuthorizationServerConfigurerAdapter;
import org.springframework.security.oauth2
        .config.annotation.web.configuration.EnableAuthorizationServer;
import org.springframework.security.oauth2
        .config.annotation.web.configurers.AuthorizationServerEndpointsConfigurer;
import org.springframework.security.oauth2
        .provider.ClientDetailsService;
//@formatter:on

@Configuration
@EnableAuthorizationServer
class AuthorizationServerConfiguration extends
    AuthorizationServerConfigurerAdapter {

    private final AuthenticationManager authenticationManager;

    private final ClientDetailsService clientDetailsService;

    @Autowired
    public AuthorizationServerConfiguration(
        AuthenticationManager authenticationManager,
        ClientDetailsService clientDetailsService) {
        this.authenticationManager = authenticationManager;
        this.clientDetailsService = clientDetailsService;
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        ❶
        clients.withClientDetails(this.clientDetailsService);
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints)
        throws Exception {
        ❷
        endpoints.authenticationManager(this.authenticationManager);
    }
}

```

❶ 配置 OAuth 客户端。

❷ 通过它的 AuthenticationManager 实例，将 Spring Security OAuth 与 Spring Security 连接起来。

保护 Greetings 资源服务器的安全

讨论了这么多内容，但什么时候才能落地呢？假设我们有一个想要保护的 REST API，希望拒绝掉那些没有有效访问令牌的访问请求。我们可以通过 @EnableResourceServer

注解来保护资源服务器。这样配置之后，Spring Security OAuth 将拒绝所有没有有效访问令牌请求。如果请求具有有效的访问令牌，则需要通过某种方式，将访问令牌转换为 Spring Security Authentication。访问令牌本身是没有意义的，它需要被翻译成与系统已认证用户相关的信息。我们可以将受保护的资源服务器指向某个获取用户信息的端点，这样 Spring Security OAuth 可以通过交换令牌，来获取有关用户的详细信息。因此在 greetings-service 资源服务器上，我们指定 security.oauth2.resource.userInfoUri = http://auth-service/UAA/user。

这个 URL 没有使用正常的主机名，而是使用了服务注册中心中的一个服务 ID。我们已经将一个能够感知服务注册和（可能）OAuth 的 RestTemplate，提取到了一个独立模块 security-autoconfiguration 的共享自动配置类中。该库只包含一个自动配置类，TokenRelayAutoConfiguration，它为我们配置了一个能够使用客户端负载均衡器的 RestTemplate。示例 8-28 展示了这个配置。

示例8-28 TokenRelayAutoConfiguration.java

```
package relay;

import feign.RequestInterceptor;

//@formatter:on
import org.springframework.boot.autoconfigure
    .condition.ConditionalOnBean;
import org.springframework.boot.autoconfigure
    .condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure
    .condition.ConditionalOnWebApplication;
import org.springframework.boot.autoconfigure
    .security.oauth2.resource.UserInfoRestTemplateFactory;
//@formatter:off

import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.Profile;
import org.springframework.http.HttpHeaders;

//@formatter:on
import org.springframework.security
    .oauth2.client.OAuth2ClientContext;
import org.springframework.security
    .oauth2.client.OAuth2RestTemplate;
import org.springframework.security
    .oauth2.client.filter.OAuth2ClientContextFilter;
import org.springframework.security
    .oauth2.config.annotation.web.configuration.EnableResourceServer;
```

```

import org.springframework.web.client.RestTemplate;
//@formatter:off

@Configuration
@ConditionalOnWebApplication
@ConditionalOnClass(EnableResourceServer.class)
public class TokenRelayAutoConfiguration {

    public static final String SECURE_PROFILE = "secure";

    @Configuration
    @Profile("!" + SECURE_PROFILE)
    public static class RestTemplateConfiguration {

        ❶
        @Bean
        @LoadBalanced
        RestTemplate simpleRestTemplate() {
            return new RestTemplate();
        }
    }

    @Configuration
    @Profile(SECURE_PROFILE)
    public static class SecureRestTemplateConfiguration {

        ❷
        @Bean
        @Lazy
        @LoadBalanced
        OAuth2RestTemplate anOAuth2RestTemplate( UserInfoRestTemplateFactory factory) {
            return factory.getUserInfoRestTemplate();
        }
    }

    @Configuration
    @Profile(SECURE_PROFILE)
    @ConditionalOnClass(RequestInterceptor.class)
    @ConditionalOnBean(OAuth2ClientContextFilter.class)
    public static class FeignAutoConfiguration {

        ❸
        @Bean
        RequestInterceptor requestInterceptor(OAuth2ClientContext clientContext ) {
            return requestTemplate -> requestTemplate.header(HttpHeaders.AUTHORIZATION,
                clientContext.getAccessToken().getTokenType() + ' '
                + clientContext.getAccessToken().getValue());
        }
    }
}

```

- ❶ 在不以 secure 结尾的 profile 文件中，安装一个普通的、支持负载均衡的 RestTemplate。

- ② 安装一个支持负载均衡的 `RestTemplate`，它也会在 `secure profile` 文件中传播一个 OAuth 令牌（如果存在的话）。
- ③ 在多个 Feign 调用中传播 OAuth 访问令牌。我们稍后再讨论。

最后，我们必须定义用户信息端点 `/uaa/user`。当一个请求被发往受保护的 REST 服务时，Spring Cloud Security 配置的过滤器会查看请求中的访问令牌，然后将令牌翻译成客户端和用户信息。令牌是不透明的，它本身不包含任何信息，它必须被变成可操作的信息。当然，这个信息的主体会从授权服务器变成另一个主体。在 Facebook 上，信息中可能会包含 Facebook 用户及其社交图谱的信息。在 GitHub 上，它可能包含 GitHub 用户及其代码提交的信息。许多端点都暴露了一个名为 `name` 的 JSON 属性。

如果我们添加一个 `@EnableResourceServer` 注解，将授权服务器配置为一个资源服务器，那么对于那些处理带访问令牌请求的 Spring MVC 处理器方法来说，Spring Security OAuth 会自动为它们提供一个 `java.security.Principal` 对象。Spring Security OAuth 还会将 `java.security.Principal` 对象转换为资源服务器可以使用的 JSON 格式。

下面我们来修改授权服务器，并创建一个将令牌转换为 `Principal` 对象的端点（如示例 8-29 所示）。

示例8-29 `PrincipalRestController.java`

```
package auth;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.security.Principal;

@RestController
class PrincipalRestController {

    ①
    @RequestMapping("/user")
    Principal principal(Principal p) {
        return p;
    }

}
```

- ① 当一个带有令牌的请求进入时，将 `Principal` 对象返回给请求者。

授权服务器会在上下文路径 `/uaa` 和端口 9191 上启动（如示例 8-30 所示）。

示例8-30 bootstrap.properties

```
spring.application.name=auth-service
1
server.context-path=/uaa
security.sessions=if_required
logging.level.org.springframework.security=DEBUG
spring.jpa.hibernate.ddl-auto=create
spring.jpa.generate-ddl=true
```

❶ spring.application.name 会作为应用程序在服务注册中心注册的标识。

有了这一切，我们应该能够生成访问令牌了。我们通过测试用例，展示了如何使用密码授权的方式，向授权服务器发送一个请求，从而生成有效的访问令牌。在密码授权中，我们从访问令牌转换出了用户名和密码。这是证明 OAuth 授权服务器工作正常的最简单的方法。

示例 8-31 展示了如何使用 curl 命令测试。

示例8-31 使用curl命令，通过密码模式来获取访问令牌

```
curl \
-X POST \
-H"authorization: Basic aHrtbDU6cGFzc3dvcmQ=" \
-F"password=spring" \
-F"client_secret=password" \
-F"client_id=html5" \
-F"username=jlong" \
-F"grant_type=password" \
-F"scope=openid" \
http://localhost:9191/uaa/oauth/token
```

当我们运行 curl 命令，并将结果传递给 JSON 格式化方法 (json_pp) 时，我们会得到如下的响应信息（如示例 8-32 所示）。

示例8-32 从OAuth2授权服务器返回的JSON响应

```
{
  "scope" : "openid",
  "expires_in" : 40222,
  "token_type" : "bearer",
  "refresh_token" : "12164df0-12a6-43d9-b631-8418aec28612",
  "access_token" : "6815d559-784c-496a-b50e-b8c91eb17ffd"
}
```

access_token 是关键要素。有了它，我们可以尝试调用一个安全的资源服务器。我们断开与资源服务器的连接，使用新创建的访问令牌来与它通信。

如果我们能够根据有效的客户端 ID、客户端密钥、用户名和密码生成一个有效的访问令牌，那么我们就能够创建一个访问边缘服务数据的 JavaScript 客户端，然后再调用 `greetings-service`。首先要确认已经保护了边缘服务（`greetings-service`）的安全。我们将示例 8-33 中的配置添加到边缘服务的代码库中。应该通过 `secure profile` 文件来启动边缘服务。

示例8-33 OAuthResourceConfiguration.java

```
package greetings;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
//@formatter:off
import org.springframework.security.oauth2.config.annotation
    .web.configuration.EnableOAuth2Client;
import org.springframework.security.oauth2.config.annotation
    .web.configuration.EnableResourceServer;
//@formatter:on

@Configuration
1
@Profile("secure")
2
@EnableResourceServer
3
@EnableOAuth2Client
class OAuthResourceConfiguration {
}
```

- ❶ 只有使用 `secure profile` 启动服务时，配置才处于激活状态。
- ❷ `@EnableResourceServer` 将拒绝它认为没有通过身份认证的请求。
- ❸ `@OpenOAuth2Client` 也会永久保存从请求中看到的任何令牌，从而允许当前节点作为另一个受保护节点的客户端。

可以将已通过身份认证的 `java.security.Principal` 对象，注入 Spring MVC 控制器方法中。这里我们重写了之前的 `greetings` 端点，重新定义了已认证用户的返回结果（如示例 8-34 所示）。它只有在使用 `secure profile` 运行 `greetings-service` 时才会存在。如果端点没有运行成功，请指定该 `profile` 并重新启动 `greetings-service` 服务。

示例8-34 SecureGreetingsRestController.java

```
package greetings;

import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import java.security.Principal;
import java.util.Collections;
import java.util.Map;

@Profile("secure")
@RestController
@RequestMapping(method = RequestMethod.GET, value = "/greet/{name}")
public class SecureGreetingsRestController {

    @RequestMapping
    Map<String, String> hi(@PathVariable String name, Principal p) {
        return Collections.singletonMap("greeting",
            "Hello, " + name + " from " + p.getName() + "!");
    }
}
```

可以使用刚刚生成的访问令牌,向 greetings-service 提交一个请求(如示例 8-35 所示)。

示例8-35 curl命令返回的安全结果

```
curl -H"authorization: bearer ..."
{"greeting":"Hello, Tammie from jlong!"}
```

当你发起请求并确认正常后,修改令牌值(可以删除一个字母或更改某些字符)然后再发起一次请求,并确认该请求被拒绝。之后,再次尝试发起同样的请求,但是不指定授权头信息(删除 -H 选项和参数),并再次确认请求被拒绝。

到目前为止,我们已经证明可以保护指定的 REST API,只允许拥有有效访问令牌的请求访问。访问令牌从哪里生成呢?你肯定不希望网站的访问者通过 curl 命令来获取访问令牌,对吧?所以我们需要为用户提供一个创建令牌的流程。

创建一个受 OAuth 保护的单页面应用程序

我们回到边缘服务。到目前为止,我们已经建立了一个授权服务器,并使用 @EnableResourceServer 注解保护了 greetings-service 的 REST API,同时拒绝了那些没有有效访问令牌的请求。为了演示安全性,我们通过密码模式,向授权服务器发送了一个请

求，希望获取一个可以转换成用户名和密码的访问令牌。当然，我们的用户不会使用通过 curl 命令向授权服务器发送 HTTP 请求的方式，来登录网站。我们需要在登录流程和客户端用户界面中集成 Spring Security OAuth。当用户使用客户端 JavaScript 访问网页时，他们将被重定向到 auth-service，弹出一个登录页面以及对某些请求的授权批准。一旦批准授权，边缘服务将获得一个访问令牌，可以将其发往所有的下游服务，如果这些服务也像 greetings-service 一样受到保护，那么也可以使用令牌来标识发起请求的用户和客户端。这在 Spring Cloud Security 中称为单点登录。

我们来看一个简单的客户端应用程序，它使用 Angular.js 编写，其显示了一个受保护 REST API 的一些状态。我们将使用简化模式进行身份认证，然后重定向到授权服务器，提示输入用户名和密码，并批准所请求的范围，最后再返回到客户端 UI 界面上。在这个例子中，我们将看到 Spring Cloud Security 如何无缝完成整个过程。

我们已经保护了一个含有多个端点的边缘服务，但是我們也需要在 src/main/resources/static 目录下添加一个 index.html 和一些 JavaScript。这些资源需要对所有人都可见，我们只需要保护 /api/* 下的 REST API（我们之前通过 RestTemplate 或者 Netflix Feign 创建的）。我们将添加一个 @EnableResourceServer 注解并进行相应的配置，然后提供一个 ResourceServerConfigurerAdapter 来获得一个有效的访问令牌（如示例 8-36 所示）。

示例8-36 SecureResourceConfiguration.java

```
package greetings;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
//@formatter:off
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.oauth2.config.annotation.web
    .configuration.EnableResourceServer;
import org.springframework.security.oauth2.config.annotation.web
    .configuration.ResourceServerConfigurerAdapter;
//@formatter:on

1
@Profile("secure")
@Configuration
@EnableResourceServer
class SecureResourceConfiguration extends ResourceServerConfigurerAdapter {

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/api/**").authorizeRequests() 2
            .anyRequest().authenticated();
    }
}
```

❶ 只有 secure profile 文件处于激活状态，该配置才有效。

❷ 只有发往 /api/* 的请求是安全的，其他一切请求都是不安全的。

我们需要为客户端配置单点登录和注销的功能。我们的应用程序会暴露一个 /login 端点、一些 JavaScript 资源和一个主页 /index.html，它们会在有人访问 / 的时候被加载。在边缘服务中，我们不仅仅拒绝了无效访问令牌请求；我们还将重定向到登录界面，启动获取一个访问令牌的流程。可以使用 @EnableOAuth2Sso 注解来安装一个身份认证过滤器和一个身份认证入口点，从而搭建一个本地端点（比如 /login），用来触发边缘服务和认证服务器之间的身份认证。我们来看一下 SSO 配置，并指定哪些端点应该通过 WebSecurityConfigurerAdapter 来触发 SSO 流程（如示例 8-37 所示）。

示例8-37 SsoConfiguration.java

```
package greetings;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

// @formatter:off
import org.springframework.boot.autoconfigure
    .security.oauth2.client.EnableOAuth2Sso;
import org.springframework.security.config.annotation
    .web.builders.HttpSecurity;
import org.springframework.security.config.annotation
    .web.configuration.WebSecurityConfigurerAdapter;
// @formatter:on

import org.springframework.security.web.csrf.CookieCsrfTokenRepository;

❶
@Profile("sso")
❷
@Configuration
@EnableOAuth2Sso
class SsoConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // @formatter:off
        http.antMatcher("/**").authorizeRequests() ❸
            .antMatchers("/", "/app.js", "/login**", "/webjars/**")
            .permitAll().anyRequest()
            .authenticated().and().logout().logoutSuccessUrl("/").permitAll()
            .and().csrf()
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
        // @formatter:on
    }
}
```


- ❶ 只有当 sso profile 文件处于激活状态时，该配置才有效。
- ❷ @ EnableOAuth2Sso 会注册所有触发 SSO 流程的必要机制。
- ❸ 我们希望保护除以下特定资源以外的所有资源。

我们的边缘服务是一个 OAuth 客户端。它通过强制用户登录来获取用户上下文，但是它需要将自己标识为一个特殊的客户端。我们可以在配置中特意指定它是什么客户端(如示例 8-38 所示)。

示例8-38 bootstrap-sso.properties

```
security.oauth2.client.client-id=html5 ❶
security.oauth2.client.client-secret=password

security.oauth2.client.access-token-uri=http://localhost:9191/uaa/oauth/token ❷
security.oauth2.client.user-authorization-uri=\
    http://localhost:9191/uaa/oauth/authorize

security.basic.enabled=false
```

- ❶ 确定用来获取某个用户上下文的客户端 ID 和密码。
- ❷ 将边缘服务指向 OAuth 授权服务器。

这样就完成了在边缘服务的服务器端获取 SSO 的整个流程。我们来编写一个 Angular.js 客户端控制器认证一下，它会试图读取受保护的信息，如果尚未通过身份认证，则会弹出登录界面。示例 8-39 给出了 Angular.js 应用程序的 HTML 模板，及其(唯一的)控制器 home 代码。

示例8-39 index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8"/>
  <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
  <title>Edge Service</title>
  <meta name="description" content=""/>
  <meta name="viewport" content="width=device-width"/>
  <base href="/" />
  <script type="text/javascript"
    src="/webjars/jquery/jquery.min.js"></script>
  <script type="text/javascript"
    src="/webjars/bootstrap/js/bootstrap.min.js"></script>
  <script type="text/javascript"
    src="/webjars/angularjs/angular.min.js"></script>
```

```

</head>

<body ng-app="app" ng-controller="home as home">

<div class="container" ng-show="!home.authenticated">
    <a href="/login">Login </a>
</div>

<div class="container" ng-show="home.authenticated">

    ❶
    Logged in as:
    <b><span ng-bind="home.user"></span></b> <br/>

    Token:
    <b><span ng-bind="home.token"></span> </b><br/>

    Greeting from Zuul Route: <b>
    <span ng-bind="home.greetingFromZuulRoute"></span></b> <br/>

    Greeting from Edge Service (Feign):
    <b><span ng-bind="home.greetingFromEdgeService"></span></b><br/>
</div>

    ❷
<script type="text/javascript" src="app.js"></script>
</body>
</html>

```

- ❶ 会显示身份认证信息，并且访问一个同一节点上受保护的 REST 端点，以显示受保护的数据。
- ❷ JavaScript 的逻辑位于另一个文件 app.js 中。

该模板会打开一个面板，显示出通过 JavaScript 代码获取和绑定的信息。JavaScript 代码的关键在于，它会调用本地节点上一个受保护的端点 /user。如果请求有一个访问令牌，它会成功；如果没有，则会初始化身份认证流程。

端点 /user 只是获取通过身份认证用户的信息，同之前 auth-service 服务的 /uaa/user 端点作用相同。由于 edge-service 使用了 @EnableResourceServer 注解，因此 /user 端点会返回 Principal 对象，供 JavaScript 应用程序使用。示例 8-40 展示了边缘服务中 /user 端点的代码（尽管你会认为它是来自 auth-service 的代码）。

示例8-40 PrincipalRestController

```

package greetings;

import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

```



```
import java.security.Principal;
```

```
@Profile("secure")
```

```
@RestController
```

```
class PrincipalRestController {
```

```
    @RequestMapping("/user")
```

```
    public Principal user(Principal principal) {
```

```
        return principal;
```

```
    }
```

```
}
```

我们来看一下 Angular.js (JavaScript) 应用程序的逻辑 (如示例 8-41 所示)。

示例8-41 app.js

```
var app = angular.module("app", []);
```

❶

```
app.factory('oauth', function () {  
    return {details: null, name: null, token: null};  
});
```

```
app.run(['$http', '$rootScope', 'oauth', function ($http, $rootScope, oauth) {
```

```
    $http.get("/user").success(function (data) {
```

```
        oauth.details = data.userAuthentication.details;
```

```
        oauth.name = oauth.details.name;
```

```
        oauth.token = data.details.tokenValue;
```

❷

```
$http.defaults.headers.common['Authorization'] = 'bearer ' + oauth.token;
```

❸

```
$rootScope.$broadcast('auth-event', oauth.token);
```

```
});
```

```
}});
```

```
app.controller("home", function ($http, $rootScope, oauth) {
```

```
    var self = this;
```

```
    self.authenticated = false;
```

❹

```
$rootScope.$on('auth-event', function (evt, ctx) {
```

```
    self.user = oauth.details.name;
```

```
    self.token = oauth.token;
```

```
    self.authenticated = true;
```

```

var name = window.prompt('who would you like to greet?');

⑤
$http.get('/greetings-service/greet/' + name)
  .success(function (greetingData) {
    self.greetingFromZuulRoute = greetingData.greeting;
  })
  .error(function (e) {
    console.log('oops!' + JSON.stringify(e));
  });

⑥
$http.get('/lets/greet/' + name)
  .success(function (greetingData) {
    self.greetingFromEdgeService = greetingData.greeting;
  })
  .error(function (e) {
    console.log('oops!' + JSON.stringify(e));
  });
});
});

```

- ❶ 为了让应用程序工作，需要一个单例对象来存储认证的细节信息。
- ❷ 应用程序加载时会调用 `app.run` 中的代码，所以其适合进行初始化操作。这里会调用受保护的 `/user` 端点来读取身份认证信息。未经身份认证的请求会被重定向到 `auth-service`，然后要求登录并接受授权。`authorization-service` 会重定向到这个页面，此次调用成功后，我们就可以为 Angular.js 所发起的所有 Ajax 调用（通过 `$http` 客户端），设置一个 `Authorization` 头信息的默认值。
- ❸ 向所有关注获取认证信息的组件，发布一个事件。
- ❹ 在这个 Angular.js 应用程序中，我们提供了一个控制器 `home`，它会将四个字段绑定到模板 `index.html` 中。
- ❺ 为了确保所有组件工作正常，Angular.js 客户端还通过 Zuul 代理和 Feign API 来调用边缘服务的端点。

如果你按照顺序运行 `service-registry`、`auth-service`、`greetings-service`（使用 `secure profile`）和 `edge-service`（使用 `secure` 和 `sso profile`）这几个服务，那么应该能够访问边缘服务，并且会被重定向到 `auth-service`。我们来看看这个流程顺序是怎样的：

1. 访问 8082 端口上的边缘服务，立即被重定向到 `auth-service`（如图 8-3 所示）。

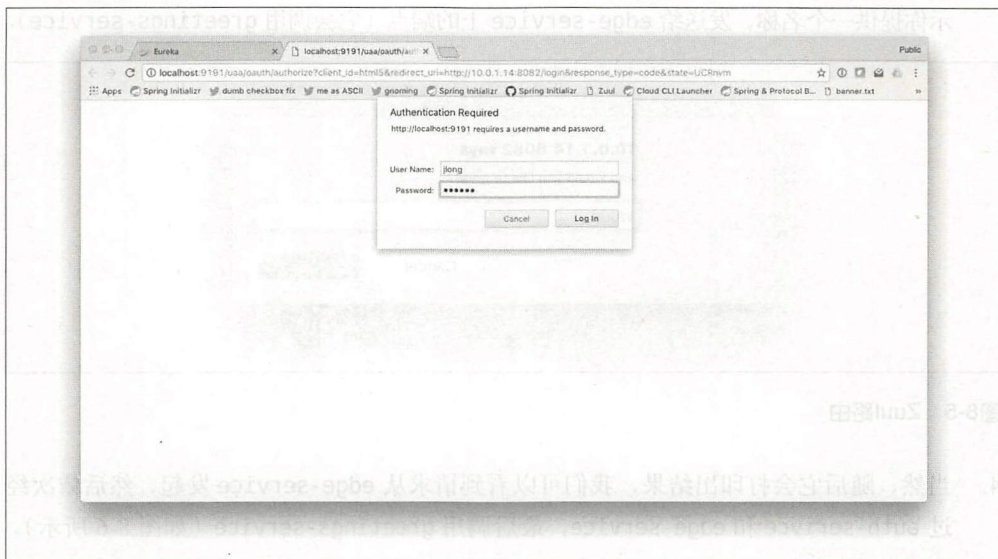


图8-3 访问边缘服务

2. 在这里你需要批准所请求的范围（如图 8-4 所示）。

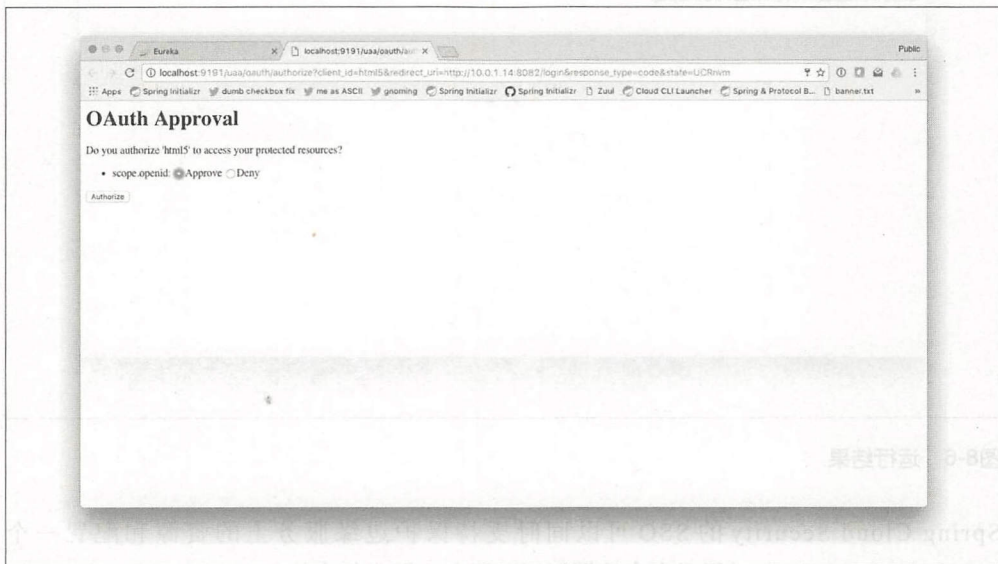


图8-4 批准请求的范围

3. 然后你将被重定向到应用程序，在那里你可以再次进行尝试（如图 8-5 所示）。这次它可以正常工作了，因为边缘服务已经获得了一个有效的访问令牌，所以它会提

示你提供一个名称，发送给 edge-service 上的端点（它会调用 greetings-service）。

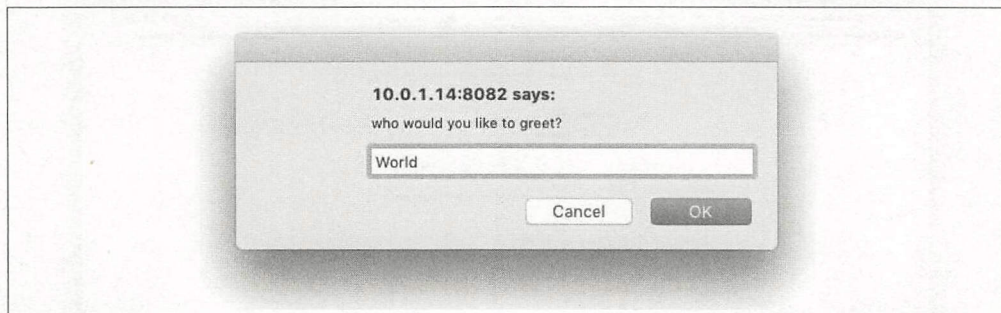


图8-5 Zuul路由

4. 当然，随后它会打印出结果，我们可以看到请求从 edge-service 发起，然后依次经过 auth-service 和 edge-service，最后调用 greetings-service（如图 8-6 所示）。

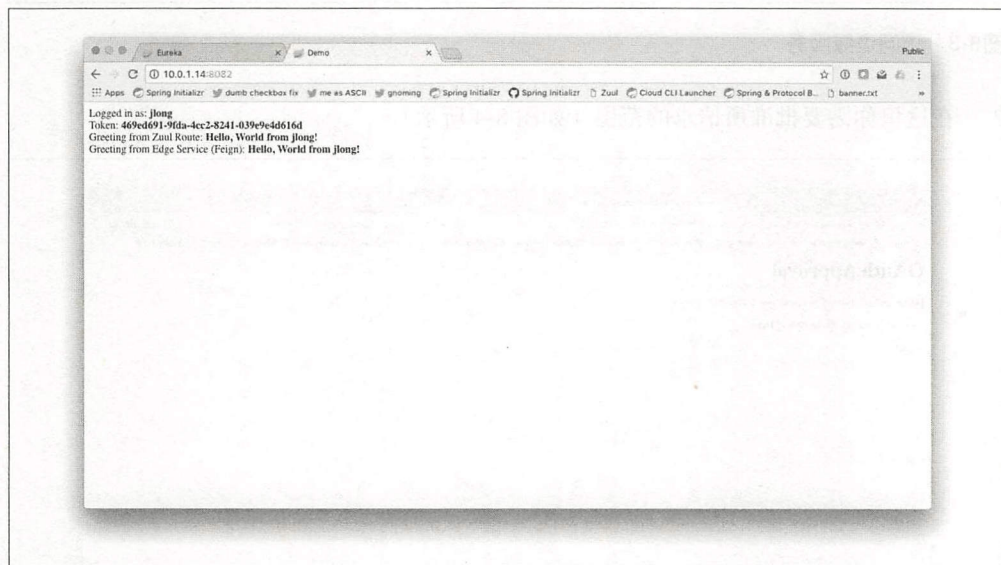


图8-6 运行结果

Spring Cloud Security 的 SSO 可以同时支持保护边缘服务上的资源和配置一个 OAuthRestTemplate（用于安全地调用下游服务，即本例中的 greetings-service）。edge-service 既是受保护的资源，也是访问另一个受保护资源的客户端。edge-service 服务和 greetings-service 一样，也在类路径中配置了我们之前提到的 edge-security-autoconfiguration 模块。

总结

在本章中，我们了解了在构建一个有效的边缘服务时会遇到的许多问题。虽然我们在本章讨论的大部分内容，都应该安排独立章节对它们进行专门、深入的讨论，但是我们这里仅关注如何将富客户端（HTML5 客户端、iOS 客户端、Android 客户端等）与微服务系统集成的问题。

我们讨论了如何有效、快速地构建客户端，以及如何借助 `RestTemplate` 以及 `Netflix Feign` 等工具，来简化与下游服务的通信工作。我们讨论了如何整合客户端负载均衡。

我们开发了一个 OAuth 身份认证服务，并将身份认证工作委派给一个支持 JPA 的 `AuthenticationProvider` 实现。也可以将身份认证工作委托给其他兼容 OAuth 标准的身份认证服务。你的授权服务器可以作为一个或多个其他下游授权服务（例如 Facebook 或者 GitHub）的门面（Facade）。本章中的 `social-auth-service` 示例就是这样做的。虽然它并不完善，但是如果你想往这条路上发展，它应该能够给你带来一些灵感。

边缘服务，至少从概念来说，与许多其他技术和模式非常相似，例如 SOFEA（面向服务的前端应用程序）、BFF（服务于前端的后端）和 API 网关中都有类似的概念。无论你怎么称呼它，这些服务都应该尽可能精简业务逻辑，且主要服务于如何集成更多的客户端及服务，同时保持足够的灵活性。

数据整合

本章将介绍在构建可扩展的云原生应用程序时的数据管理问题。我们先回顾一些熟悉的领域数据建模方法，然后看看 Spring Data 项目为了管理仓库如何向外暴露存储库。我们还将看到一些微服务示例，使用 Spring Data 项目来管理对数据源的独占数据访问。

数据建模

良好的数据模型可以有效地传达软件中的业务需求，就像图 9-1 中的领域模型一样。领域模型可以用来表达业务领域中的重要问题。Eric Evans 在其开创性的著作 *Domain-Driven Design: Tackling Complexity in the Heart of Software*（领域驱动设计：软件核心复杂性应对之道）（Addison-Wesley 出版社出版）中首次提出了领域建模。这也是迄今为止最成功的领域建模技术。

Evans 指出业务领域专家和 IT 组织中的软件工程师应当使用明确的术语进行沟通，通过明确描述软件中的对象和模块的方式来推广领域驱动设计的概念。

领域驱动设计旨在解决的问题是软件设计中的复杂性。Evans 以“软件核心复杂性应对之道”作为书的小标题，意思就是说，技术人员应该把重心放在通过解决业务底层模型的复杂性来使软件更易于构建和维护上。这种复杂性存在于公司为服务客户而创建的业务流程和功能之间；领域模型反映出这种复杂性，并为技术人员提供了一种能够使企业更好地编写软件的语言。

考虑下面的例子。如果工程师在一个软件模块中将客户（customer）命名为用户（user），那么工程师就会被迫使用含糊不清的术语，这可能与业务领域的专家的意思完全不同。这是一个客户（customer）账户（account）管理的用户（user）故事：

作为客户（customer），我希望能够创建用户（user），该用户能够管理一组账户（account）。

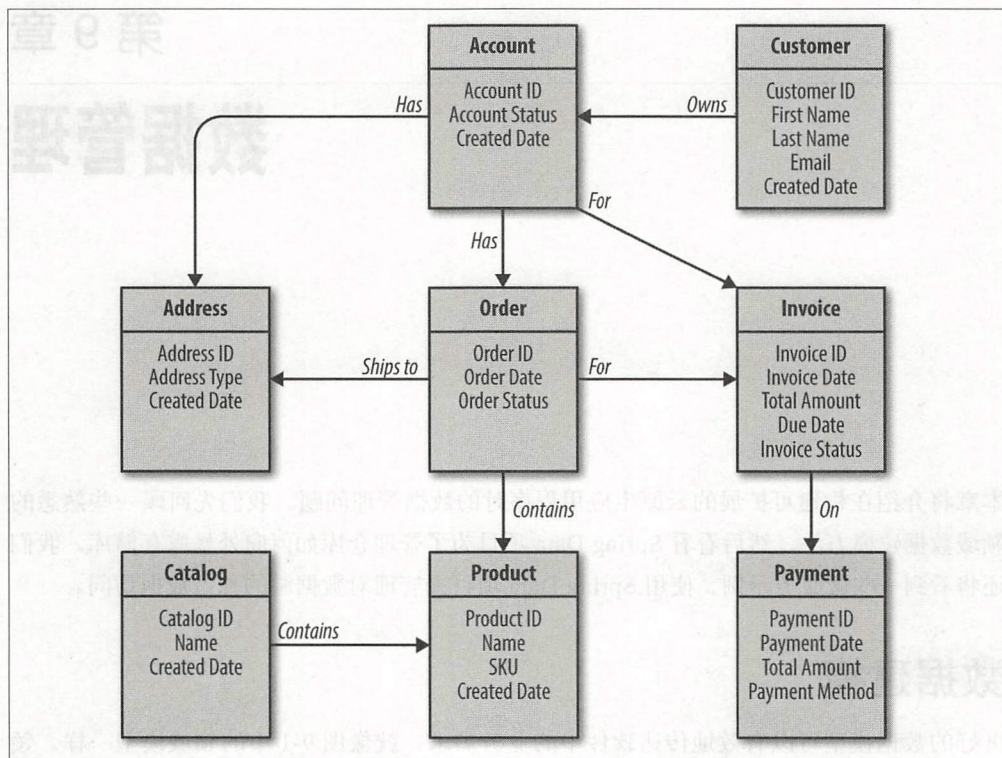


图9-1 表示领域类之间关系的领域模型

这个用户故事是由领域专家编写的。这个故事描述了一个账户管理用例，明确地区分客户和用户，但是并不能确定用户是否也可以是客户。

如果不区分这些领域模型，工程师可能只是在代码中简单地创建一个名为 `User` 的领域类。在后来的版本中，企业可能会澄清“客户”与“用户”的不同。这可能会带来高昂的代价，导致新功能的开发验证滞后，软件发布延期。

大多数数据仓库仅专注于提供最佳数据存储方案，而对可用性关注不足。我们认为，任何能够使业务技术人员更容易达成共识的事情都值得追求。数据仓库不光是用来存储字节的，更是用来描述业务领域与实体的关系的。我们的目标是最大限度地提高在特定技术上的投资（包括时间和金钱）回报，同时最大限度地减少绑定到特定数据存储的代码，这些代码不会改进我们的业务目标。这就是为什么我们要使用 `Spring` 和 `Spring Data` 的原因。

关系数据库管理系统（RDBMS）

几十年来，关系数据库模型一直是事务型数据存储解决方案的主要组成部分。随着技术

发展到云原生架构，市场上开始要求与 RDBMS 具有相同事务保证的其他类型的数据模型。SQL 数据库已经成为万金油，无所不能。与所有持久性技术相比，当前的 SQL 数据库并不能提供最强大的事务性需求。更讽刺的是，以表 (table) 的方式来描述关系并不是最好的方式。SQL 数据库支持二进制数据，但不是特别好用。许多数据库支持地理信息，但这不一定是吸引人的功能。SQL 数据库可以处理大量的读取事务，但是通常只有在模式 (schema) 设计者对模型进行了非规范化处理之后，才能保证一致性，获得更高的可用性。简而言之，关系数据库不会过时，也值得专门研究。如果你正在使用 RDBMS，那么在 JPA 实现之上的 JPA，如 Hibernate，在将记录映射到 RDBMS 模式或从 RDBMS 模式映射到记录时，就显得非常合适。JPA（以及对象关系映射器 ORM）可以让开发者更多地关注领域模型，而不是关注 RDBMS 本身的细微差别。这将牺牲（一些）控制以易于演进。

NoSQL

NoSQL 数据库提供了多种数据模型，其中包含很多优化，以满足特定场景的需求。这在微服务中十分有用，因为我们可以使用 NoSQL 数据库的特殊特征来分解领域模型的有界上下文。访问数据是微服务 API 完成的，因此客户端不必关心使用了什么技术。

Polyglot Persistence（异构持久性）的术语是由 Martin Fowler 提出的，用于描述使用多种数据库模型的体系结构。建立在单个关系数据库上的应用程序可以慢慢分解成使用 NoSQL 数据库的微服务，具体过程取决于应用程序用例。对于那些需要使用更复杂解决方案的新领域、新品牌、新项目特别有用，并且可以快速交付。由于 NoSQL 数据库提供了针对用例优化过的特定数据模型，因此用户很少需要为了解决某个问题而修改关系数据库模型。

Spring Data

Spring Data (<http://projects.spring.io/spring-data>) 是一个开源的伞式项目，它提供了一个大家熟悉的抽象，即在与数据存储进行交互的同时保留其数据库模型的特性。有十几个 Spring Data 项目（官方的）涵盖了一系列流行的数据库，包括 RDBMS 和 NoSQL 数据模型。Spring Data 模块支持 JDBC、JPA、MongoDB、Neo4J、Redis、Elasticsearch、Solr、Gemfire、Cassandra 和 Couchbase 等。



JPA 和 Spring Data JPA 涵盖了所有流行的 RDBMS 供应商，包括 Oracle、MySQL、PostgreSQL、SQL Server、H2、HSQL、Derby 等。

Spring Data 应用程序的结构

在开始使用 Spring Data 之前，有必要先了解一下数据访问层的模式。我们首先了解一下数据存储交互的基本组成。

域类

第一个组件是实体类，代表域模型中的一个对象。域类（*domain class*）是一个基本的类，其作为域数据的模型。域类由一组专用字段组成，并根据域模型（如示例 9-1 所示）的设计，使用 `public getter` 和 `setter` 公开其内容。

示例 9-1 表示用户模型的基本域类

```
public class User { ❶
    private Long id;
    private String firstName; ❷
    private String lastName;
    private String email;
```

- ❶ 类名代表了域模型中的概念、名词或实体。
- ❷ 私有字段被映射到本地数据存储区中的数据对象属性。



域类表示映射到应用程序的数据存储中的数据对象（如表）的实体。

库

在 Spring Data 应用程序中访问数据的主要方法是存储库（*repository*）。Eric Evans 在他的著作 *Domain-Driven Design*（领域驱动设计）中介绍了存储库的概念。Spring 框架长期以来一直支持构造型注释 `@Repository`，它除了将类标记为 `@Component` 之外，还向 Spring 发出信号，告知 bean 可能会抛出与持久化技术相关的异常，并且这些异常应该被标准化到 Spring 提供的层次结构中。通过这种方式，数据消费者只需要关心违反约束条件的一类异常。

Spring Data 提供更进一步的支持。Spring Data 可以提供用户定义接口的实现，其数据管理方法的签名是按照 Spring Data 约定来定义的。你可以定义自己的接口方法，或者扩展一些方便的接口定义。这些接口中最基本的实现是 `CrudRepository`。



CRUD 是 *CREATE*、*READ*、*UPDATE* 和 *DELETE* 几个单词首字母的缩写词。这四个动词描述了任何持久存储技术中数据的基本管理功能。不巧的是，HTTP 协议使用稍微不同的动词来管理资源（如网络事务），它们是 *POST*、*GET*、*PATCH* 和 *DELETE*。

`CrudRepository` 支持常见的 *CRUD* 操作，其提供了两个信息：域类类型和 *ID* 类型。我们可以使用 `CrudRepository` 接口来创建 `User` 域类的存储库（如示例 9-2 所示）。

示例9-2 `User`域类的Spring Data存储库

```
public interface UserRepository extends CrudRepository<User, Long> {  
}
```

我们不需要实现 `UserRepository` 接口。Spring Data 将提供一个实现契约的 bean，我们可以根据需要将其注入其他 Spring bean。

为领域数据组织 Java 包

在构建微服务时需要重点考虑软件包的组织风格。建议你按照自己感觉最舒适的方式来组织类和包，但是这里我们会提供一种适用于 Spring Data 和 Spring Boot 构建微服务的包组织模式。

首先需要确定当前的领域模型中有界的上下文是什么。考虑图 9-2 中的领域模型。

该图表示一个领域模型，在 `customer` 上下文和 `Order` 上下文之间有明确的界限。假设我们想创建两个微服务来涵盖每个有界的上下文。我们需要组织软件包，以便稍后可以轻松地针对领域概念迁移域类和存储库。为了做好重构的准备，我们可以将管理单个领域类的域类和存储库分组到一个包中。图 9-3 和图 9-4 显示了在这个域中代码的结构。

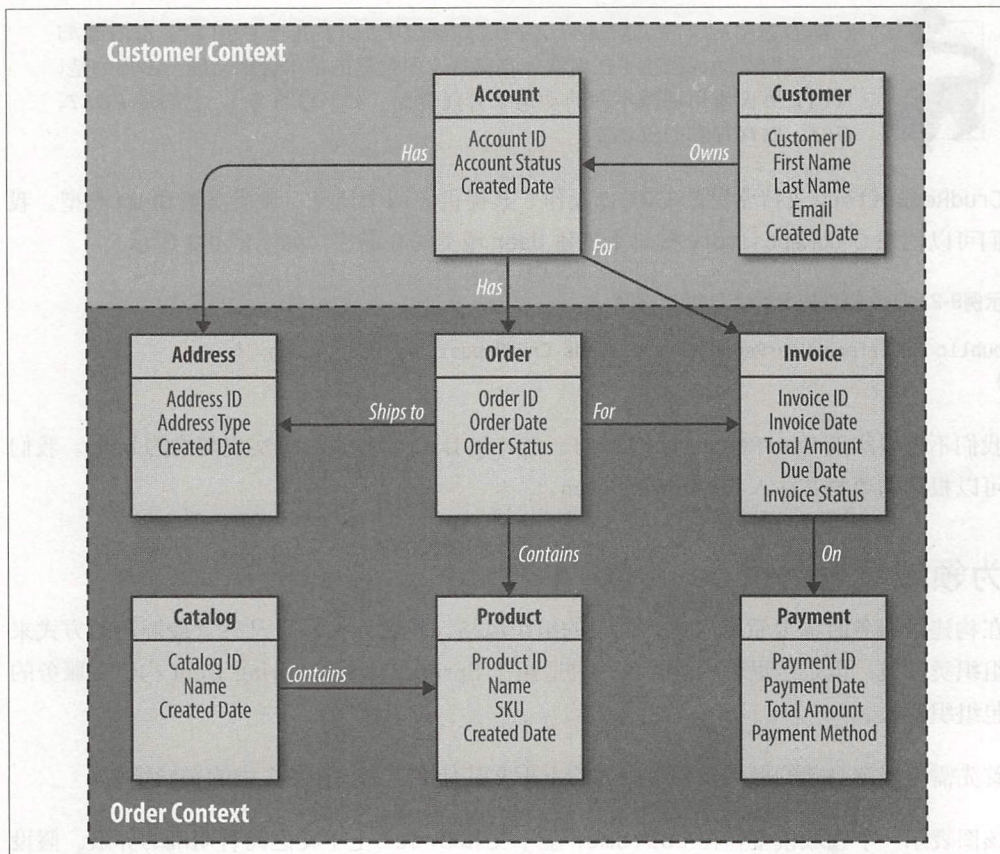


图9-2 具有两个有界上下文的领域模型

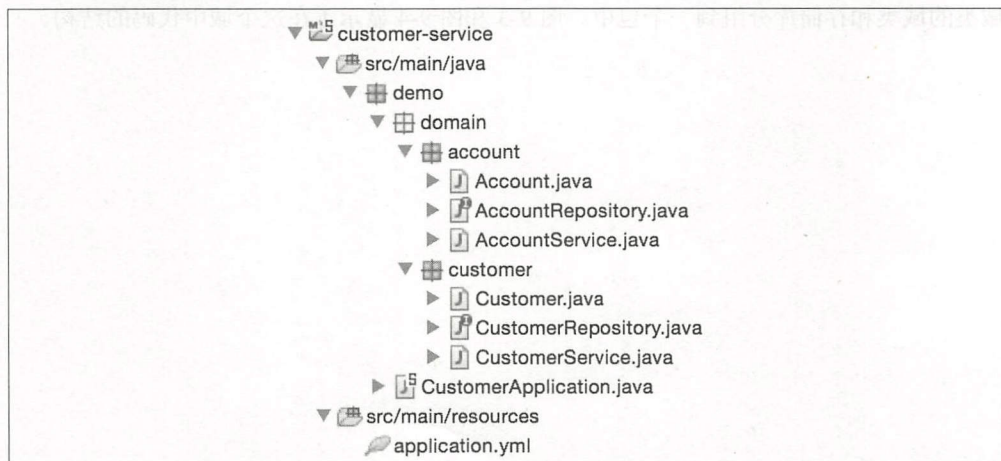


图9-3 Customer上下文的包结构

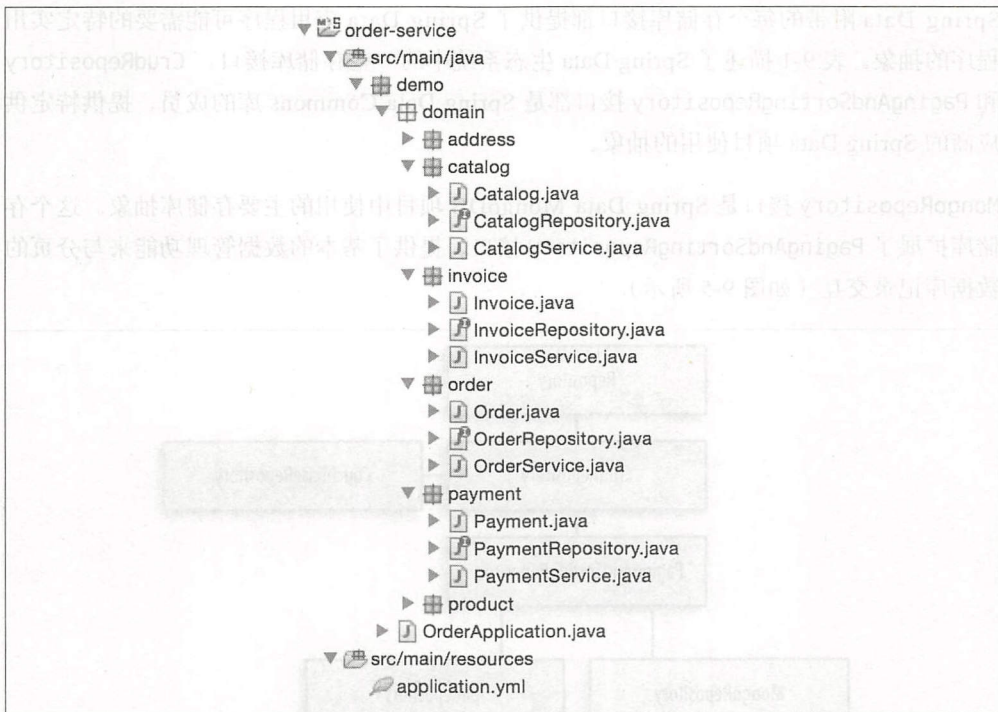


图9-4 Order上下文的包结构

支持的存储库

Spring Data 项目支持扩展 Repository 接口的多种存储库，表 9-1 只显示了其中一些。

表9-1 常见Spring Data存储库

存储库类型	Spring Data 项目	目的
Repository	Spring Data Commons	为 Spring Data 存储库提供核心抽象
CrudRepository	Spring Data Commons	扩展 Repository 并为基本的 CRUD 操作添加实用程序
PagingAndSortingRepository	Spring Data Commons	扩展 CrudRepository 并添加用于分页和排序记录的实用程序
JpaRepository	Spring Data JPA	扩展 PagingAndSortingRepository 并为 JPA 和 RDBMS 数据库模型添加实用程序
MongoRepository	Spring Data MongoDB	扩展 PagingAndSortingRepository 并添加用于管理 MongoDB 文档的实用程序
CouchbaseRepository	Spring Data Couchbase	扩展 CrudRepository 并添加用于管理 Couchbase 文档的实用程序

Spring Data 附带的每个存储库接口都提供了 Spring Data 应用程序可能需要的特定实用程序的抽象。表 9-1 描述了 Spring Data 生态系统中的一些存储库接口。CrudRepository 和 PagingAndSortingRepository 接口都是 Spring Data Commons 库的成员，提供特定供应商的 Spring Data 项目使用的抽象。

MongoRepository 接口是 Spring Data MongoDB 项目中使用的主要存储库抽象。这个存储库扩展了 PagingAndSortingRepository 接口，提供了基本的数据管理功能来与分页的数据库记录交互（如图 9-5 所示）。

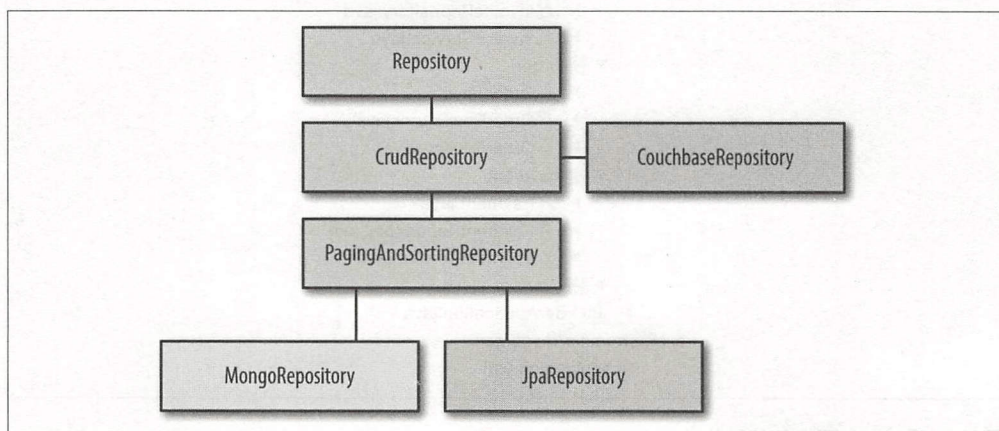


图9-5 Spring Data Commons中的三个基础知识库提供对特定于供应商的数据存储的基本数据访问

使用 JDBC 访问 RDBMS 数据

Spring Boot 使用一组默认设置引导应用程序的依赖关系。我们至少需要配置一个 SQL 的 DataSource、JDBC 和 JPA。如果 Spring Boot 在类路径中找到一个众所周知的 DataSource 驱动程序，那么它将配置一个到本地数据存储的连接。在我们的例子中，我们将使用 MySQL，将 mysql:mysql-connector-java 添加到类路径中。如果我们将 spring-boot-starter-data-jpa 启动程序添加到类路径中，它将配置 JPA。JPA 启动程序将自动配置 javax.sql.DataSource 指向 MySQL，并将其用作 JPA 配置的数据源。

你可以很方便地通过 JDBC 和 JPA 属性来定制 DataSource。理想情况下，诸如数据源之类的敏感信息应该位于应用程序代码之外。有关配置的更多信息，请参阅我们在第 3 章中对配置的讨论。在特定开发环境的配置文件下，在代码库的配置中保存较不敏感的仅限于开发时使用的默认值也是合理的。示例 9-3 给出了一个例子。

示例9-3 application.yml

```
spring:
  profiles:
    active: development
---
spring:
  profiles: development
  jpa:
    database: MYSQL
  datasource: ❶
    url: jdbc:mysql://localhost/test
    username: dbuser
    password: dbpass
```

- ❶ spring.datasource 属性块是配置 RDBMS 特定连接细节的地方。



默认情况下，Spring Boot 为 JDBC DataSource 自动配置一个连接池。连接池默认基于 Apache Tomcat DBCP，但可以通过将类库添加到类路径来配置 Commons DBCP 或 Hikari CP。

Spring 的 JDBC 支持

JdbcTemplate 是 Spring 框架中的经典数据访问机制之一。JdbcTemplate 提供了支持 JDBC 规范的基于 SQL 的关系数据库的管理功能。本节将介绍如何使用 JdbcTemplate。



JdbcTemplate 很可能是模板设计模式中最普遍和最著名的实现，因为从 Spring 早期开始，它就已经存在了。Spring Data 项目包括许多支持其他数据存储的模板实现。

现在，我们可以使用 application.yml 来为 development 配置文件执行 SQL 查询，将其作为数据源配置。我们创建一个 User 表，并使用 JdbcTemplate 来与这些记录交互（如示例 9-4 所示）。

示例9-4 使用原始SQL和JdbcTemplate处理数据源

```
package demo;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
```

```

import org.springframework.boot.CommandLineRunner;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Component;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

@Component
class JdbcCommandLineRunner implements CommandLineRunner {

    private final Logger log = LoggerFactory.getLogger(getClass());

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    JdbcCommandLineRunner(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public void run(String... strings) throws Exception {

        ❶
        jdbcTemplate.execute("DROP TABLE user IF EXISTS");
        jdbcTemplate
            .execute("CREATE TABLE user( id serial, first_name " +
                " VARCHAR(255), last_name VARCHAR(255), email VARCHAR(255))");

        ❷
        List<Object[]> userRecords = Stream
            .of("Michael Hunger michael.hunger@jexp.de",
                "Bridget Kromhout bridget@outlook.com", "Kenny Bastani kbastani@yahoo.com",
                "Josh Long jlong@hotmail.com").map(name -> name.split(" "))
            .collect(Collectors.toList());

        jdbcTemplate
            .batchUpdate(
                "INSERT INTO user(first_name, last_name, email) VALUES (?, ?, ?)",
                userRecords);

        ❸
        RowMapper<User> userRowMapper = (rs, rowNum) -> new User(rs.getLong("id"),
            rs.getString("first_name"), rs.getString("last_name"), rs.getString("email"));

        List<User> users = jdbcTemplate.query(
            "SELECT id, first_name, last_name, email FROM user WHERE first_name = ?",

```



```

        userRowMapper, "Michael");

users.forEach(user -> log.info(user.toString()));
}
}

```

- ❶ 创建 schema。
- ❷ 调出一些示例记录，并使用 `JdbcTemplate` 中方便的 `batchUpdate` 功能将它们写入数据库。
- ❸ 通过将它们映射到对象来访问每个记录，然后记录这些对象的结果。

希望这些代码简洁到使人一目了然。如果你使用过 JDBC API，那么应该就会知道这里的大部分代码与创建 `DataSources`、`Connections` 和 `Statements`，通过结果集 `cursing`，事务管理，处理可能永远不会出现的异常，如果出现任何问题就回滚有关。`JdbcTemplate` 消除了这一切事情，让我们专注于手头问题的本质。当我们运行一个查询时，它会自动创建一个语句，运行查询，访问每条记录，并给我们提供的 `RowMapper <T>` 实现一个回调来转换结果。

运行这个例子，你会看到控制台上出现结果 Michael Hunger（如示例 9-5 所示）。

示例9-5 使用原始SQL和JdbcTemplate与数据源进行交互

```
User(id=1, firstName=Michael, lastName=Hunger, email=michael.hunger@jexp.de)
```

这个例子看起来微不足道，只是用来强调 Spring 模板对象的核心概念。我们将在 Spring Data 中看到这种支持。为了简单起见，我们运行 SQL 语句来创建 Java 代码中的模式和表。然而，还有一个更简洁的选择。按照惯例，Spring 将在应用程序启动时评估两个文件：`src/main/resources/schema.sql` 和 `src/main/resources/data.sql`。将任何模式 DDL（数据定义语言），例如用于创建用户和表的语句放在 `schema.sql` 中。我们可以将 *Bridget*，*Michael* 等的样例记录放到 `data.sql` 文件中。然后，只需要最后几行代码来处理来自数据源记录的查询和映射。

Spring Data 示例

我们已经探索了一些 Spring Data 的基本概念：域类和存储库。下面我们将更深入地了解使用不同技术的项目示例。接下来我们将介绍下面这些 Spring Data 项目的数据服务例子。

- Spring Data JPA（MySQL）
- Spring Data MongoDB

- Spring Data Neo4j
- Spring Data Redis



请记住，我们还将使用编译时注释处理器 Project Lombok (<https://projectlombok.org/>) 来减少代码库中的样板代码量。@Data 告诉 Lombok 产生访问器 (accessor) 和增变器 (mutator)，一个 toString 方法和一个 equals/hashCode 方法。@NoArgs-Constructor 告诉 Lombok 生成一个没有参数的构造函数。@AllArgsConstructor 告诉 Lombok 生成一个构造函数，它接受类中的每个字段。我们可以自由地生成自己的构造函数，或者覆盖任何访问器 / 增变器，即使存在这些注释。

我们将把这些数据示例集的示例域应用到一个在线店面上。我们将使用描述企业资源计划 (ERP) 应用程序的领域模型。我们的每一项服务都将扩展到一个特定的领域。

假设该示例项目的在线商店应用程序由虚拟的 *Monolithic* 有限公司使用，这是一家正在将其在线商店的应用程序开发迁移到云端的服装品牌公司。

Monolithic 有限公司正陷于麻烦之中，他们想让开发团队在其现有的网上商店中部署新功能。受到更多现代软件开发方法的启发，这家创业公司已决定将其产品重新命名为 *Cloud Native Clothing*。

让我们来看看 *Cloud Native Clothing* 用于构建其新的在线商店的领域模型 (见图 9-6)。

从 *Cloud Native Clothing* 的领域模型可以看到，有几个有界的上下文。图中每个有界的上下文都被标记为一个上下文。根据 DDD 中提出的原则，有界的上下文是将业务领域的一部分模块化为单独的统一模型的一种方式。每个有界的上下文可以有一组无关的概念，同时明确共享概念之间的联系。我们决定创建一个可以阐明业务领域概念的模式，同时使用 Spring Data 存储库阐明应用程序是如何构建的。

图中的每个域类都被表示为一个圆。每个圆也代表一个 Spring 数据仓库。在该图中的上下文中，任何一个需要在有界上下文中公开查询能力的任何域类是必需的；因此当使用这种模型时，我们总是认为每个圆都代表一个域类和存储库。存储库之间的相互关系表明了每个领域类中的某种程度的连通性。

通过定向关系连接的概念指示域类之间的显式关系。通过虚线连接的概念表示通过存储库连接推断的连通性。虚线表示应该有一个存储库查询作为域概念之间的连接桥。桥应该只在两个存储库之间存在的域类的共享概念的上下文中有有效。例如，*Account* 桥接 *Order* 和 *Address* (如图 9-7 所示)。

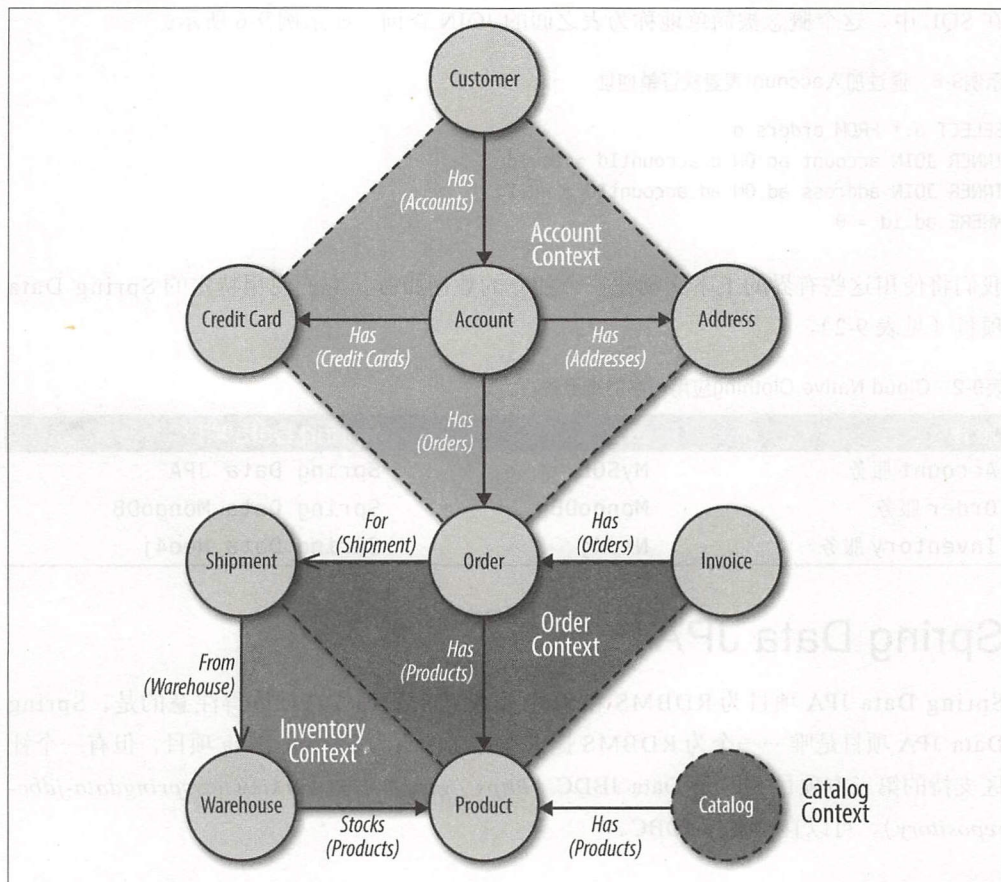


图9-6 Cloud Native Clothing的领域模型

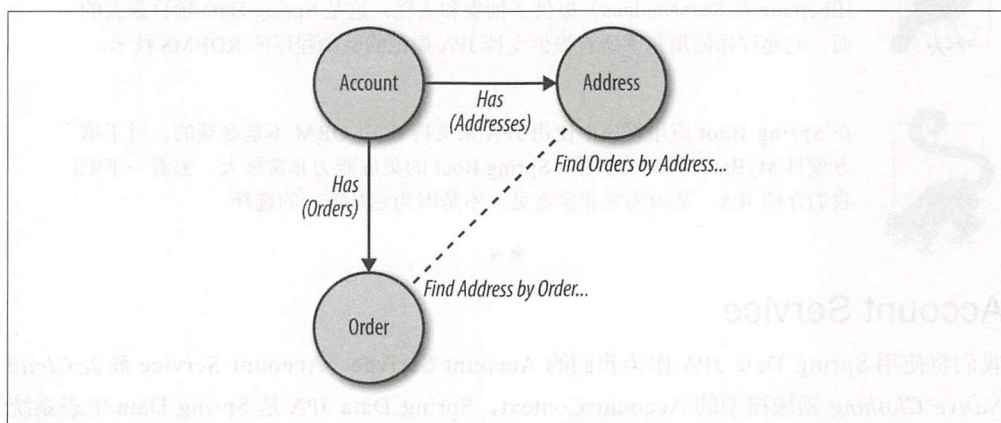


图9-7 推断的关系可以通过共享的域类查找

在 SQL 中，这个概念被简单地称为表之间的 JOIN 查询，如示例 9-6 所示。

示例9-6 通过加入account表查找订单地址

```
SELECT o.* FROM orders o
INNER JOIN account ac ON o.accountId = ac.Id
INNER JOIN address ad ON ad.accountId = ac.Id
WHERE ad.id = 0
```

我们将使用这些有界的上下文创建三个独立的数据服务，它们使用特定的 Spring Data 项目（见表 9-2）。

表9-2 Cloud Native Clothing应用程序的微服务

服务	数据源	Spring Data 项目
Account 服务	MySQL	Spring Data JPA
Order 服务	MongoDB	Spring Data MongoDB
Inventory 服务	Neo4j	Spring Data Neo4j

Spring Data JPA

Spring Data JPA 项目为 RDBMS 和 SQL 提供数据管理支持。值得注意的是，Spring Data JPA 项目是唯一一个为 RDBMS 提供存储库抽象的 Spring Data 项目，但有一个社区支持的第三方项目 Spring Data JDBC (<https://github.com/nurkiewicz/springdata-jdbc-repository>)，可以直接支持 JDBC。



JPA 代表 Java Persistent API，它是在 JCP (Java Community Process) 中首次描述和引入的 JSR 220 的一部分。JPA 为特定于供应商的 ORM 技术（如 Hibernate 和 DataNucleus）提供了抽象和实现。这是 Spring Data 项目强大的一面，它允许你使用几乎所有提供支持 JPA 规范的驱动程序的 RDBMS 技术。



在 Spring Boot 应用程序中使用 JPA 来支持 SQL ORM 不是必须的。对于第三方项目 MyBatis 和 JOOQ 等，Spring Boot 的集成能力非常强大。去看一下吧！我们介绍 JPA，是因为它非常常见，不是因为它是唯一的选择。

Account Service

我们将使用 Spring Data JPA 作为我们的 Account Service。Account Service 涵盖 *Cloud Native Clothing* 领域模型的 Account Context。Spring Data JPA 是 Spring Data 生态系统中的一个特殊项目。Spring Data JPA 提供了一个单独的库，其可以很好地与多个 (JDBC-capable) 数据源搭配使用，其他 Spring Data 项目使用一种数据源。这样做的一个好处是，

我们可以使用嵌入式数据源进行测试，并在其他地方切换到更全面的数据库。我们将利用 Spring 的 *profiles* 来实现清晰的分离。

在本节中，我们将介绍为 *Cloud Native Clothing* 的 *Account Service* 设置 Spring Data JPA 应用程序的主要问题。

使用不同数据源的配置文件

我们在 Maven 构建中添加了三个依赖关系来启用我们的 JPA 域：

- org.springframework.boot:spring-boot-starter-data-jpa
- com.h2database:h2
- mysql:mysql-connector-java

现在我们有 Spring Data JPA 项目的框架，并在类路径中附带 MySQL 和 H2 数据库驱动程序，我们需要配置应用程序使其能够在正确的上下文中使用这两个数据库。MySQL 数据库在应用程序的外部运行，我们将在运行应用程序时远程连接它。运行集成测试时，我们只想使用嵌入式 H2 数据库。Spring 提供了一种配置应用程序的方法，使其能够根据配置文件在运行时选择不同的配置来运行。在 *application.yml* 文件中，你可以创建一个 *development profile* 和一个 *test profile*（如示例 9-7 所示）。

示例9-7 配置应用程序进行开发和测试

```
spring:
  profiles:
    active: development ❶
---
spring:
  profiles: development ❷
  jpa:
    show_sql: false
    database: MYSQL
    generate-ddl: true
  datasource:
    url: jdbc:mysql://192.168.99.100:3306/dev
    username: root
    password: dbpass
---
spring:
  profiles: test ❸
  jpa:
    show_sql: false
    database: H2
  datasource:
```

```
url: jdbc:h2:mem:testdb;DB_CLOSE_ON_EXIT=FALSE
```

- ❶ 运行应用程序时，这是默认情况下的 active profile。
- ❷ 这是 development profile 的配置。
- ❸ 这是 test profile 的配置。

现在我们为开发和测试配置定义了两个单独的 profile，我们需要配置集成测试来使用 test profile。由于我们已经将应用程序属性中的 active profile 默认值设置为 development，因此集成测试将尝试使用 development profile，我们不希望这样做。为了解决这个问题，只需要在我们的集成测试环境中覆盖 active profile 即可。

通过这个设置，我们只需要在 Spring 集成测试中指定 @ActiveProfiles，指向 test profile。当运行 Account Service 的集成测试时，将使用 H2 嵌入式内存数据库。通过使用嵌入式数据库，我们可以在任何构建环境中运行测试，而不用担心必须连接到外部依赖项。

使用 JPA 描述 Account Service 的域

我们将为 Account Service 的域类创建 JPA 实体。JPA 类是一个注释的域类，它将被映射到关系数据库（在这种情况下是 MySQL 或 H2）中的表。

在我们的 *Cloud Native Clothing* 领域模型中有 4 个有界的上下文。在 Account Context 中，有 5 个存储库：*Account*、*Address*、*Order*、*CreditCard* 和 *Customer*（见图 9-8）。

创建 *Account Service* 的下一步是构建域类。每个 Spring Data 项目创建一个域类的过程可能稍有不同。对于 Spring Data JPA，在工具箱中有几个工具，可以用来注释部分域类以匹配 *Account Context* 中的域模型。

现在，我们使用表中描述的 JPA 注释为 Account Service 创建每个域类作为 JPA 实体类。我们需要为 *Account Context* 中的每个概念创建域类，其中包括以下对象（见图 9-9）。

- Account
- Customer
- CreditCard
- Address

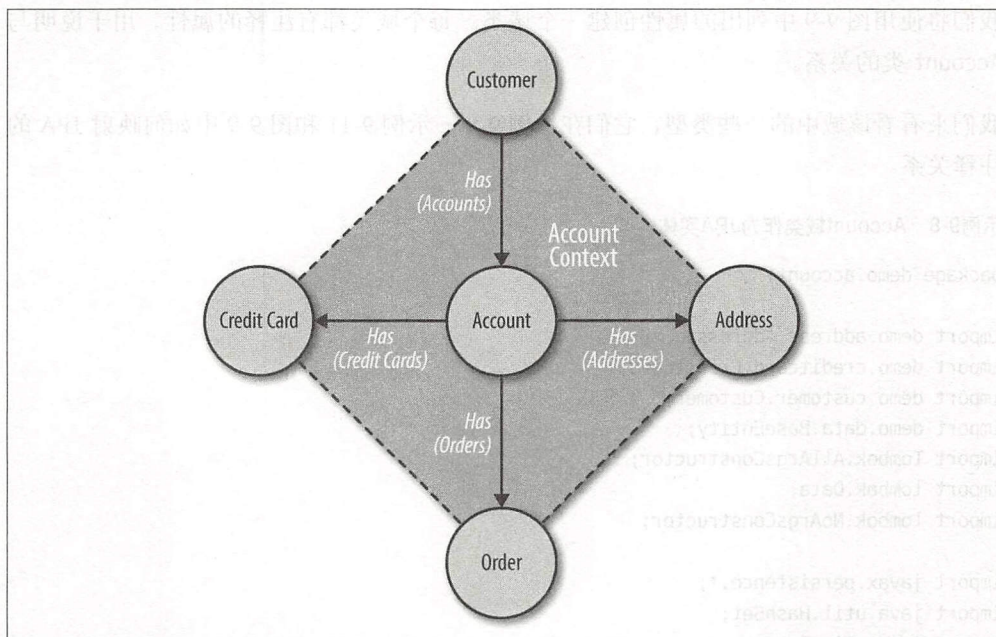


图9-8 Account Service的Account Context

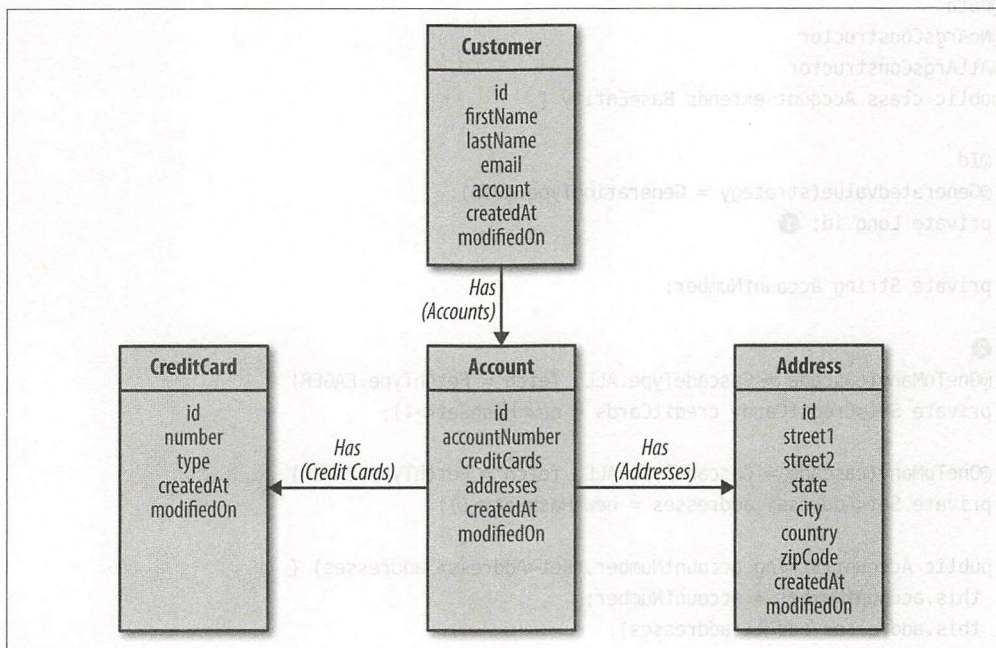


图9-9 Account Context的实体关系图

我们将使用图 9-9 中列出的属性创建一个域类。每个域类都有注释的属性，用于说明与 Account 类的关系。

我们来看看该域中的一些类型，它们在示例 9-8 ~ 示例 9-11 和图 9-9 中如何映射 JPA 的注释关系。

示例9-8 Account域类作为JPA实体

```
package demo.account;

import demo.address.Address;
import demo.creditcard.CreditCard;
import demo.customer.Customer;
import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Account extends BaseEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id; ❶

    private String accountNumber;

    ❷
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<CreditCard> creditCards = new HashSet<>();

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<Address> addresses = new HashSet<>();

    public Account(String accountNumber, Set<Address> addresses) {
        this.accountNumber = accountNumber;
        this.addresses.addAll(addresses);
    }

    public Account(String accountNumber) {
```



```

        this.accountNumber = accountNumber;
    }
}

```

❶ @GeneratedValue 将增加 @Id 字段的唯一 ID 的值。在这里，我们已经（冗余地和明确地）指定主键应该使用自动递增的值。

❷ @OneToMany 注解描述了一个到 JPA 实体的 FK 关系。

示例9-9 Customer域类作为JPA实体

```
package demo.customer;
```

```

import demo.account.Account;
import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NoArgsConstructor;

```

```
import javax.persistence.*;
```

```
@Entity
```

```
@Data
```

```
@NoArgsConstructor
```

```
@AllArgsConstructor
```

```
public class Customer extends BaseEntity {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    private Long id;
```

```
    private String firstName;
```

```
    private String lastName;
```

```
    private String email;
```

```
    @OneToOne(cascade = CascadeType.ALL)
```

```
    private Account account;
```

```
    public Customer(String firstName, String lastName, String email,
```

```
        Account account) {
```

```
        this.firstName = firstName;
```

```
        this.lastName = lastName;
```

```
        this.email = email;
```

```

        this.account = account;
    }
}

```

示例9-10 Address域类作为JPA实体

```

package demo.address;

import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Address extends BaseEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String street1;

    private String street2;

    private String state;

    private String city;

    private String country;

    private Integer zipCode;

    @Enumerated(EnumType.STRING)
    private AddressType addressType;

    public Address(String street1, String street2, String state, String city,
        String country, AddressType addressType, Integer zipCode) {
        this.street1 = street1;
        this.street2 = street2;
        this.state = state;
        this.city = city;
    }
}

```



```

    this.country = country;
    this.addressType = addressType;
    this.zipCode = zipCode;
}
}

```

示例9-11 CreditCard域类作为JPA实体

```

package demo.creditcard;

import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;

import javax.persistence.*;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class CreditCard extends BaseEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String number;

    @Enumerated(EnumType.STRING)
    private CreditCardType type;

    public CreditCard(String number, CreditCardType type) {
        this.number = number;
        this.type = type;
    }
}

```

请注意，我们已经为各种类中的 `@OneToMany` 注解提供了一些参数（如示例 9-12 所示）。

示例9-12 我们可以在 `@OneToMany` 注解中定制映射

```

@OneToMany (cascade = CascadeType.ALL, fetch = FetchType.EAGER)
private Set <Address> address = new HashSet <> ();

```

- CascadeType.ALL 表示提交事务级联到所有引用的 JPA 实体。
- FetchType.EAGER 表示关系会自动填充。

@OneToMany 注解描述了由 JPA 实体类管理的表之间的外键关系。在与 Address 有 Account 关系的情况下，有一个基数 1..* 或一对多关系。

使用 JPA 进行审计

我们还可以对 JPA 类进行审计，以记录记录的创建日期以及上次修改时间。这是每个 JPA 实体都会继承的 BaseEntity 类的责任。我们来看看 BaseEntity 类(如示例 9-13 所示)。

示例9-13 BaseEntity类提供JPA审计

```
package demo.data;

import lombok.Data;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;
import org.springframework.data.jpa.domain.support.AuditingEntityListener;

import javax.persistence.EntityListeners;
import javax.persistence.MappedSuperclass;

@Data
@MappedSuperclass
1 @EntityListeners(AuditingEntityListener.class)
2
public class BaseEntity {

    @CreatedDate
    private Long createdAt; 3

    @LastModifiedDate
    private Long lastModified; 4

}
```

- 1 这个注解指定了一个 JPA 实体继承的超类。
- 2 这指定了一个审计回调监听器，它将观察 JPA 操作的生命周期。
- 3 创建记录时，此注释将保存时间戳。
- 4 记录更新时，此注释将应用更新的时间戳。

最后一步是为 Spring Boot 应用程序启用 JPA 审计。我们需要将 `@EnableJpaAuditing` 注解应用到我们的应用程序的配置类中。

现在我们已经知道了如何使用映射的超类创建 JPA 实体类，同理，我们也可以对 `Account Context` 中的其他域类执行相同的操作。现在我们创建存储库，以便管理 `Account Service` 的数据。

`Account` 和 `customer` 实体是聚合实体，当 `Account` 和 `Customer` 实体存活和死亡时，所有其他实体都存活和死亡。因此，我们将创建存储库来管理它们（如示例 9-14 和示例 9-15 所示）。

示例9-14 AccountRepository的定义

```
package demo.account;

import org.springframework.data.repository.PagingAndSortingRepository;

public interface AccountRepository extends
    PagingAndSortingRepository<Account, Long> {

}
```

示例9-15 CustomerRepository的定义

```
package demo.customer;

import org.springframework.data.repository.PagingAndSortingRepository;
import java.util.Optional;

public interface CustomerRepository extends
    PagingAndSortingRepository<Customer, Long> {

    ❶
    Optional<Customer> findByEmailContaining(String email);
}
```

- ❶ 在这个仓库中，我们按照惯例自定义了一个查找方法，该方法返回 `Optional<Customer>`，其中包含一个 `Customer` 或者什么都不包含。在幕后，这个查找方法变成了使用 JPA-QL 的查询。我们可以使用 `@Query` 注解来覆盖精确的查询。



集成测试

使用实体，确认一切按预期工作（如示例 9-16 所示）。

示例9-16 AccountApplicationTests的定义

```
package service;

import com.mysql.jdbc.AssertionFailedException;
import demo.AccountApplication;
import demo.account.Account;
import demo.address.Address;
import demo.address.AddressType;
import demo.creditcard.CreditCard;
import demo.creditcard.CreditCardType;
import demo.customer.Customer;
import demo.customer.CustomerRepository;
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.Collection;
import java.util.Optional;

import static demo.creditcard.CreditCardType.VISA;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = AccountApplication.class)
@ActiveProfiles(profiles = "test")
public class AccountApplicationTests {

    @Autowired
    private CustomerRepository customerRepository;

    @Test
    public void customerTest() {
        Account account = new Account("12345");
        Customer customer = new Customer("Jane", "Doe", "jane.doe@gmail.com", account);
        CreditCard creditCard = new CreditCard("1234567890", VISA);
        customer.getAccount().getCreditCards().add(creditCard);

        String street1 = "1600 Pennsylvania Ave NW";
        Address address = new Address(street1, null, "DC", "Washington",
            "United States", AddressType.SHIPPING, 20500);
```




```

customer.getAccount().getAddresses().add(address);

customer = customerRepository.save(customer);
Customer persistedResult = customerRepository.findOne(customer.getId());
Assert.assertNotNull(persistedResult.getAccount()); ❶
Assert.assertNotNull(persistedResult.getCreatedAt());
Assert.assertNotNull(persistedResult.getLastModified()); ❷

Assert.assertTrue(persistedResult.getAccount().getAddresses().stream()
    .anyMatch(add -> add.getStreet1().equalsIgnoreCase(street1))); ❸

customerRepository.findByEmailContaining(customer.getEmail()) ❹
    .orElseThrow(
        () -> new AssertionFailedException(new RuntimeException(
            "there's supposed to be a matching record!")));
}
}

```

- ❶ 成功地保存了一个记录和一对一的关系。
- ❷ 成功地保存了一个记录，审计机制已经奏效。
- ❸ 成功地保存了一个记录和一个关联。
- ❹ 在这里，通过我们的自定义查找方法来查找所有具有特定电子邮件的 Customer 实体。

Spring Data MongoDB

Spring Data MongoDB 项目为 MongoDB 提供基于存储库的数据管理。

MongoDB 是一个 NoSQL 数据库，因易用性和简化的数据模型而闻名。MongoDB 是面向文档的数据库，这意味着记录被存储为分层的 JSON-like 文档。如果将 `org.springframework.boot:spring-boot-starter-data-mongodb` 添加到应用程序的类路径中，则可以很容易地连接到 MongoDB 实例。使用 `spring.data.mongodb.host`、`spring.data.mongodb.port` 和 `spring.data.mongodb.database` 属性，可以指定连接到一个特定的实例。否则，Spring Boot 会配置到本地实例的连接。

Order Service

我们的 Order Service 将使用 *Spring Data MongoDB*，它将作为 *Cloud Native Clothing* 域模型中 Order Context 的后端服务。与上一节类似，这一节我们将创建一个 Spring Data MongoDB 应用程序，所以创建 Account Service 的大部分知识在这里也适用。





并非所有的 Spring Data 项目都支持将数据源作为嵌入式内存数据库运行，例如 *Spring Data JPA* 和 H2 或 HSQLDB。由于 MongoDB 是用 C++ 编写的，因此它不能作为 JVM 应用程序中的进程嵌入。在这一节，你需要下载并安装 MongoDB (<http://www.mongodb.org/>)。

使用 Spring Initializr 创建一个新项目，并确保在依赖关系中选择了 `org.springframework.boot:spring-boot-starter-data-mongodb`。

对于 Order Service，我们将返回 *Order Context* (*Cloud Native Clothing* 的领域模型) (见图 9-10)。

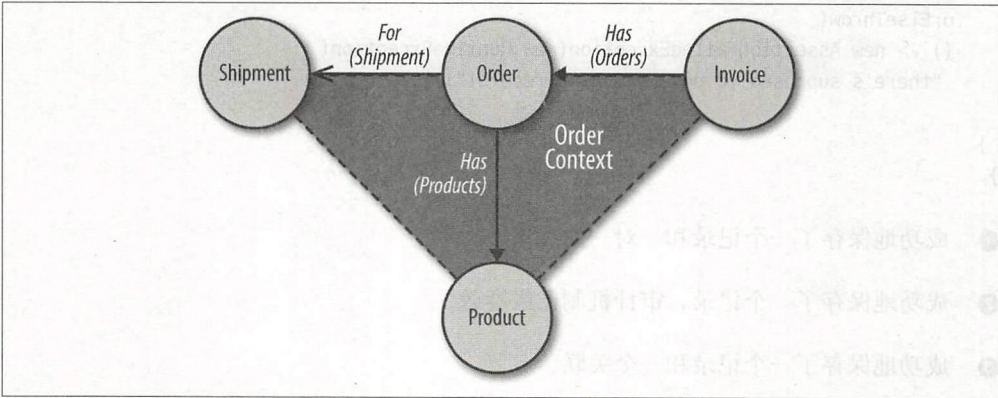


图9-10 Order Service的Order Context

在 *Order Context* 中，可以看到有 4 个领域类：*Invoice*、*Order*、*Product* 和 *Shipment*。为了理解如何构建领域类，我们来看看 Order Service 的主要用例，如表 9-3 所示。

表9-3 Order Service的主要用例

行为	域类
订单包括一组产品和数量	Order、Product
发货是将订单发送到某个地址	Shipment、Address、Order
发票是由账户生成的一组订单	Invoice、Order、Account

此表提供了比我们通常在企业应用程序中看到的更简单的主要用例集。这三个用例描述了 Order Service 中最显著的行为。请注意，域类彼此交互。在领域驱动设计中，把用例分解成描述 actor 与域对象交互行为的语句是很重要的。

使用 MongoDB 的文档类

我们在本章前面看到，Account Service 中的每个域类都使用名为 `@Entity` 的 JPA 注



解进行注释。对于其他 Spring Data 项目，用于域类注释的模式保持不变。每个 Spring Data 项目注释的语义都是特定于数据库模型特征的。对于 Spring Data MongoDB，我们可以在类上使用 @Document 注解来表示它是 MongoDB 中文档的表示。

我们将创建的第一个文档类是 Invoice。对于我们的主要用例，我们有一个声明，可以使用它来描述应该如何构建 Invoice 域类：

- 发票是由账户生成的一组订单。

这里的目标是创建一个模型类来支持该用例的行为，不要有太多限制。当我们在 Order Service 中创建新的发票时，应该要求发票可以通过账号查找。这个账号将被返回到由 Account Service 管理的 Account 域类。我们还将提供一组订单的字段，而不是每个发票的单一订单。并为 Invoice 提供账单地址，这个地址描述了发票的发送地点（如示例 9-17 和示例 9-18 所示）。



之所以为 Order Service 使用文档数据库，是因为它是聚合存储。这很有用，因为在用户提交订单之后，订单中的订单项不应该被修改。每个订单项的产品应代表下订单时产品的状态。

示例9-17 Invoice域类作为MongoDB文档

```
package demo.invoice;

import demo.address.Address;
import demo.address.AddressType;
import demo.data.BaseEntity;
import demo.order.Order;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.ArrayList;
import java.util.List;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Document
public class Invoice extends BaseEntity {

    @Id
```



```

private String invoiceId;

private String customerId;

private List<Order> orders = new ArrayList<Order>();

private Address billingAddress;

private InvoiceStatus invoiceStatus;

public Invoice(String customerId, Address billingAddress) {
    this.customerId = customerId;
    this.billingAddress = billingAddress;
    this.billingAddress.setAddressType(AddressType.BILLING);
    this.invoiceStatus = InvoiceStatus.CREATED;
}

public void addOrder(Order order) {
    order.setAccountNumber(this.customerId);
    orders.add(order);
}
}

```

示例9-18 Invoice状态的枚举

```

package demo.invoice;

public enum InvoiceStatus {
    CREATED, SENT, PAID
}

```

我们将需要使用一个 Spring Data MongoDB 存储库来简化文档实例管理（如示例 9-19 所示）。

示例9-19 InvoiceRepository，完成一个自定义查找方法

```

package demo.invoice;

import demo.address.Address;
import org.springframework.data.repository.PagingAndSortingRepository;

public interface InvoiceRepository extends
    PagingAndSortingRepository<Invoice, String> {

    Invoice findByBillingAddress(Address address);
}

```



从 Invoice 域类可以看到，我们正在引用一组 Order 类。这里再次回顾一下为 Order 提供更多详细信息的主要用例：

- 订单包括一组产品和数量。
- 发货是将订单发送到某个地址。

这两句话描述了几个领域类的交互。为了满足第一条语句的要求，我们使用一个名为 LineItem 的新对象类来描述 Product 及其数量。每个 LineItem 都会引用一个远程引用，存储在 Inventory Service 中的 Product，我们将在后面的章节中创建它。第二个用例语句描述了一个 Shipment 的创建，它被传递给 Order 的 Address。为了满足这个声明的要求，我们将要求在 Order 类的构造函数中提供账号和送货地址。最后，提供一个状态对象，描述一个 Order 对象从初始状态变化到最终状态的状态（如示例 9-20 和示例 9-21 所示）。

示例9-20 Order域类

```
package demo.order;

import demo.address.Address;
import demo.address.AddressType;
import demo.data.BaseEntity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.ArrayList;
import java.util.List;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Document
public class Order extends BaseEntity {

    @Id
    private String orderId;

    private String accountNumber;

    private OrderStatus orderStatus;

    private List<LineItem> lineItems = new ArrayList<>();
```



```

private Address shippingAddress;

public Order(String accountNumber, Address shippingAddress) {
    this.accountNumber = accountNumber;
    this.shippingAddress = shippingAddress;
    this.shippingAddress.setAddressType(AddressType.SHIPPING);
    this.orderStatus = OrderStatus.PENDING;
}

public void addLineItem(LineItem lineItem) {
    this.lineItems.add(lineItem);
}
}

```

示例9-21 订单状态的枚举

```

package demo.order;

public enum OrderStatus {
    PENDING, CONFIRMED, SHIPPED, DELIVERED
}

```

现在我们创建一个 `LineItem` 类，它描述在产品目录中产品的引用以及产品在订购时的状态（如示例 9-22 所示）。

示例9-22 `LineItem`类

```

package demo.order;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class LineItem {

    private String name, productId;

    private Integer quantity;

    private Double price, tax;
}

```



使用 MongoDB 进行审计

使用支持审计的 MongoDB 实体。首先需要有一个基础实体，其他实体可以继承用于审计的公共字段，如示例 9-23 所示。

示例9-23 审计抽象基类

```
package demo.data;

import lombok.Data;
import org.joda.time.DateTime;

@Data
public class BaseEntity {

    private DateTime lastModified, createdAt;
}
```

然后，需要配置一个 AbstractMongoEventListener，如示例 9-24 所示——在实体生命周期更改中贡献这些值。

示例9-24 使用侦听器来审计MongoDB事务

```
package demo.data;

import org.joda.time.DateTime;
import org.springframework.data.mongodb.core.mapping.event.AbstractMongoEventListener;
import org.springframework.data.mongodb.core.mapping.event.BeforeSaveEvent;
import org.springframework.stereotype.Component;

@Component
class BeforeSaveListener extends AbstractMongoEventListener<BaseEntity> {

    @Override
    public void onBeforeSave(BeforeSaveEvent<BaseEntity> event) {

        DateTime timestamp = new DateTime();

        if (event.getSource().getCreatedAt() == null)
            event.getSource().setCreatedAt(timestamp);

        event.getSource().setLastModified(timestamp);

        super.onBeforeSave(event);
    }
}
```

集成测试

现在我们已经为 Order Service 创建了数据层，接下来我们来创建一个集成测试（如示例 9-25 所示）。我们将在测试中执行以下步骤：

- 创建一个新的 Order。
- 添加一个 LineItem 到新的 Order 中。
- 使用 OrderRepository 保存 Order。

示例9-25 MongoDB组件的集成测试

```
package orders;

import demo.OrderApplication;
import demo.address.Address;
import demo.invoice.Invoice;
import demo.invoice.InvoiceRepository;
import demo.order.LineItem;
import demo.order.Order;
import demo.order.OrderRepository;
import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = OrderApplication.class)
public class OrderApplicationTest {

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private InvoiceRepository invoiceRepository;

    @Before
    @After
    public void reset() {
        orderRepository.deleteAll();
        invoiceRepository.deleteAll();
    }

    @Test
```



```

public void orderTest() {

    Address shippingAddress = new Address("1600 Pennsylvania Ave NW", null, "DC",
        "Washington", "United States", 20500);
    Order order = new Order("12345", shippingAddress);
    order.addLineItem(new LineItem("Best. Cloud. Ever. (T-Shirt, Men's Large)",
        "SKU-24642", 1, 21.99, .06));
    order.addLineItem(new LineItem("Like a BOSH (T-Shirt, Women's Medium)",
        "SKU-34563", 3, 14.99, .06));
    order.addLineItem(new LineItem(
        "We're gonna need a bigger VM (T-Shirt, Women's Small)", "SKU-12464", 4,
        13.99, .06));
    order.addLineItem(new LineItem("cf push awesome (Hoodie, Men's Medium)",
        "SKU-64233", 2, 21.99, .06));
    order = orderRepository.save(order);
    ❶
    Assert.assertNotNull(order.getId());
    Assert.assertEquals(order.getLineItems().size(), 4);

    ❷
    Assert.assertEquals(order.getLastModified(), order.getCreatedAt());
    order = orderRepository.save(order);
    Assert.assertNotEquals(order.getLastModified(), order.getCreatedAt());

    ❸
    Address billingAddress = new Address("875 Howard St", null, "CA",
        "San Francisco", "United States", 94103);
    String accountNumber = "918273465";

    Invoice invoice = new Invoice(accountNumber, billingAddress);
    invoice.addOrder(order);
    invoice = invoiceRepository.save(invoice);
    Assert.assertEquals(invoice.getOrders().size(), 1);

    ❹
    Assert.assertEquals(invoiceRepository.findByBillingAddress(billingAddress),
        invoice);
}
}

```

- ❶ 首先，确认已经将数据写入了持久性存储。
- ❷ 然后确认审计正在工作。
- ❸ 确认嵌套关联的工作。
- ❹ 这里我们使用一个自定义的查找方法，Spring Data MongoDB 将其转换成了针对 MongoDB Invoice 文档的 BSON 查询。

Spring Data Neo4j

Spring Data Neo4j 项目为流行的 NoSQL 图数据库 Neo4j 提供基于数据库的数据管理。图数据库将数据库对象表示为节点和关系的连接图。节点和关系可以包含简单的值类型和复杂的值类型。图数据库基于图论的原理。图论背后的数学原理使得图数据库能够平均每秒遍历数百万个关系，并对大型互连数据集进行建模。乍一看，这似乎是一个关系数据库的价值主张，但它不是——试试看，在 RDBMS 中建模 Facebook 好友关系图，然后做一个 leftouter join！在图中，节点之间的关系和节点本身一样重要。关系可能具有方向，单向或双向；一个人“是另一个人的雇员”。

与 Cypher 相结合时，Neo4j 的声明性查询语言与 SQL 相似，这种连接模型非常有用。Cypher 查询语言提供使用模式查询和管理数据的能力，而 SQL 则使用严格的模式，必须在查询数据之前定义严格的模式。此外，Neo4j 提供了定义图遍历的功能，消除了 SQL 复杂的连接，这些连接使复杂的查询难以读写。

在本节中，我们将使用 *Spring Data Neo4j* 为 *Cloud Native Clothing* 的在线商店的后端创建一个 Inventory Service。以下是我们想要实现的：

- 概述 *Cloud Native Clothing* 对库存管理的要求。
- 回顾 Neo4j 如何将数据库对象作为连接图来管理。
- 为库存管理设计图数据模型。
- 将 Spring 数据域类实现为 Neo4j 节点。
- 创建存储库来管理 Neo4j 节点的域类。

Inventory Service

我们将 Inventory Service 的构建分为两部分：设计和实现。我们需要研究如何构建一个图数据模型来描述 *Cloud Native Clothing* 的在线商店的库存情况。

再看看我们的 *Cloud Native Clothing* 在线商店的例子。服装店可能有一个按季度划分的目录，它可以用来在每个季度开始时在网站上更改一组产品。并不是说每季产品目录中的产品都必须有不同。店内可能会有有一些产品全年销售。因此，目录可以有一个层次结构，例如全年提供产品的基本目录，以及每季度改变产品线的季节性目录。

我们还需要将产品连接到世界各地的仓库。这些仓库中可能有目录中的产品。我们可以跟踪产品库存，并使用这些信息创建最接近交货地点的货件。这是一个复杂的事情。为了解决这个问题，我们需要解决连接模型的复杂性问题，而不需要花太多时间在设计上。

Neo4j 为我们提供了以直观的方式建模数据的灵活性，我们可以使用本章前面部分介绍

的抽象概念将这些数据映射到 Spring Data 应用程序。

如图 9-11 所示的库存上下文显示了应用程序构建的简化视图。为了使这个应用程序符合我们前面提到的要求，考虑运输和库存管理等功能，我们需要设计一个图模型来构建域类和存储库。我们先来看 Spring Data Neo4j 如何使用 Neo4j 的属性图数据模型来管理连接的数据。

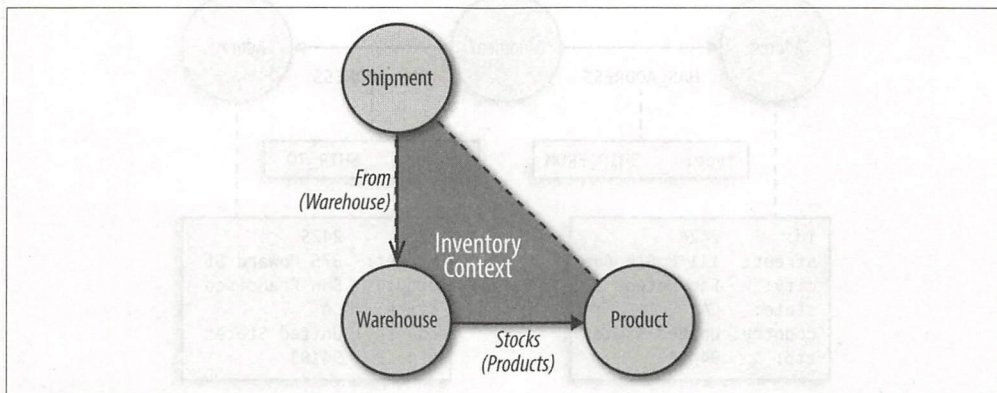


图9-11 Inventory Service的上下文

配置 Neo4j

Spring Boot 提供了 Spring Data Neo4j 项目的自动配置能力。我们可以使用一系列有关图数据库特性的功能。为了构建 Neo4j 应用程序，需要在 Maven 构建中使用 `org.springframework.boot:spring-boot-starter-data-neo4j`。你可以修改 `spring.data.neo4j.uri`、`spring.data.neo4j.username`、`spring.data.neo4j.password` 等属性将你的应用程序指向一个特定的 Neo4j 实例。我们的 Spring Boot 项目被配置为使用 Spring Data Neo4j，我们可以使用在前面例子中使用的抽象来构建一个应用程序，以管理 Neo4j 数据库中的连接数据。

使用 Neo4j 进行图数据建模

图数据建模比为 RDBMS 域数据建模更直观。在本节中，我们将介绍如何使用 Neo4j 的属性图数据模型来创建一个节点和关系图，我们将用它来构建 Spring Data 存储库和域类。首先回顾一下 Neo4j 如何将数据对象作为图模型中的实体进行管理。

Neo4j 有两个本地数据实体：节点和关系。节点用于存储键 / 值属性的映射。关系还允许你存储属性的映射。可以将简单的值类型（如 `color` 或 `gender`）作为属性存储在节点上。复杂值类型（如地址）应该表示为单独的节点。该关系用于将节点连接在一起，例如将 `address` 节点连接到 `shipment` 节点。

在图 9-12 中，可以看到，一个出货节点连接到两个地址节点。关系类型 `HAS_ADDRESS` 可以连接到多个地址。

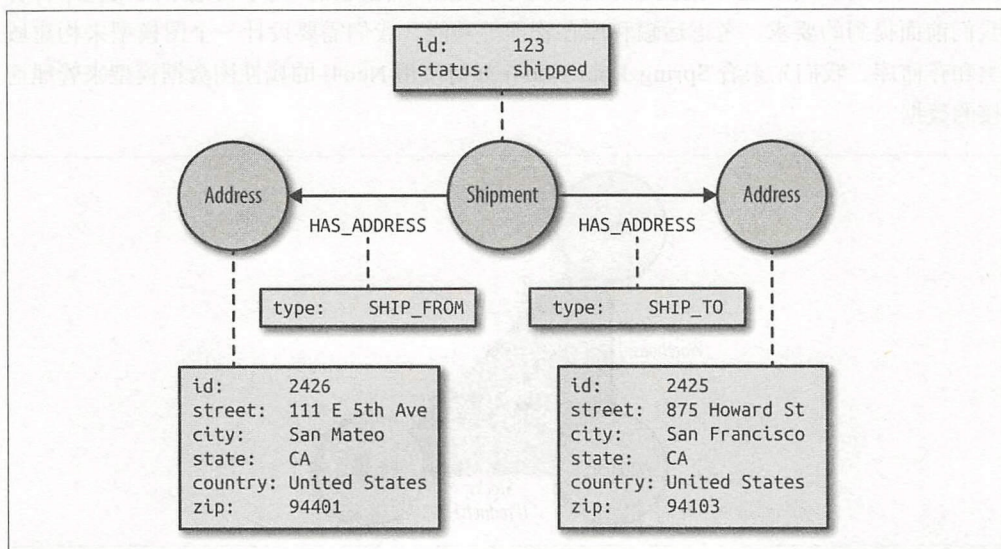


图9-12 关系的属性用于限定节点的复杂值类型

使用 Cypher，我们可以表述一个查询来获取发货的每个地址：

```
MATCH (shipment:Shipment)-[r:HAS_ADDRESS { type: "SHIP_FROM" }]->(address)
WHERE shipment.id = 123
RETURN address
```

在这个查询中，首先匹配所有的发货节点，由 `(shipment:Shipment)` 描述。Cypher 中的节点和关系使用 ASCII 艺术来将图形实体描述为模式。一个节点被圆括号封装起来，用一个冒号分隔 Cypher 应该存储结果的变量，比如 `(shipment)` 和一个描述节点属于哪个组的标签，比如 `(:Shipment)` 表示一个节点有 *Shipment* 标签。

如果使用该查询，会得到以下结果：

Detailed Query Results

Query Results

```
+-----+-----+
| address |
+-----+-----+
| Node[7]{street:"875 Howard St",country:"United States",id:2425,city:"San Francisco",zip:"94103",state:"CA"} |
| Node[6]{street:"111 E 5th Ave",country:"United States",id:2426,city:"San Mateo",zip:"94401",state:"CA"} |
+-----+-----+
2 rows
54 ms
```


我们看到，已经检索了具有 ID 123 的特定货物的两个地址。如果我们只想查询货物从哪个仓库发货呢？则可以将 Cypher 查询修改为以下内容：

```
MATCH (shipment:Shipment)-[r:HAS_ADDRESS { type: "SHIP_FROM" }]->(address)
WHERE shipment.id = 123
RETURN address
```

现在我们只得到这两个地址中的一个，如下所示：

Detailed Query Results	
Query Results	

address	
Node[6]{street:"111 E 5th Ave",country:"United States",id:2426,city:"San Mateo",zip:"94401",state:"CA"}	
-----	-----
1 row	
71 ms	

修改第一个 Cypher 查询以包含 *HAS_ADDRESS* 关系上的一个属性类型，`()-[r:HAS_ADDRESS { type: "SHIP_FROM" }]->()`，这应该等于 *SHIP_FROM*，我们能够获得发货的地址信息。

现在我们知道了 Neo4j 如何将数据模型化为图，下面我们可以创建一个描述库存微服务的节点和关系的图模型。

在图 9-13 中，可以看到，每个节点都包含有标签和节点之间的关系，这为我们提供了一个粗略的草图，其描绘了如何在 Spring Boot 应用程序中使用 Spring Data Neo4j 构建域类和存储库。

在 Spring Data Neo4j 中，Neo4j 图模型中的标签（例如图 9-13 中的节点上的标签）将成为我们的 Spring Data 存储库。我们将有以下每个节点标签的存储库，如图数据模型所示：

- Address（地址）
- Shipment（发货）
- Inventory（库存）
- Product（产品）
- Catalog（目录）
- Warehouse（仓库）

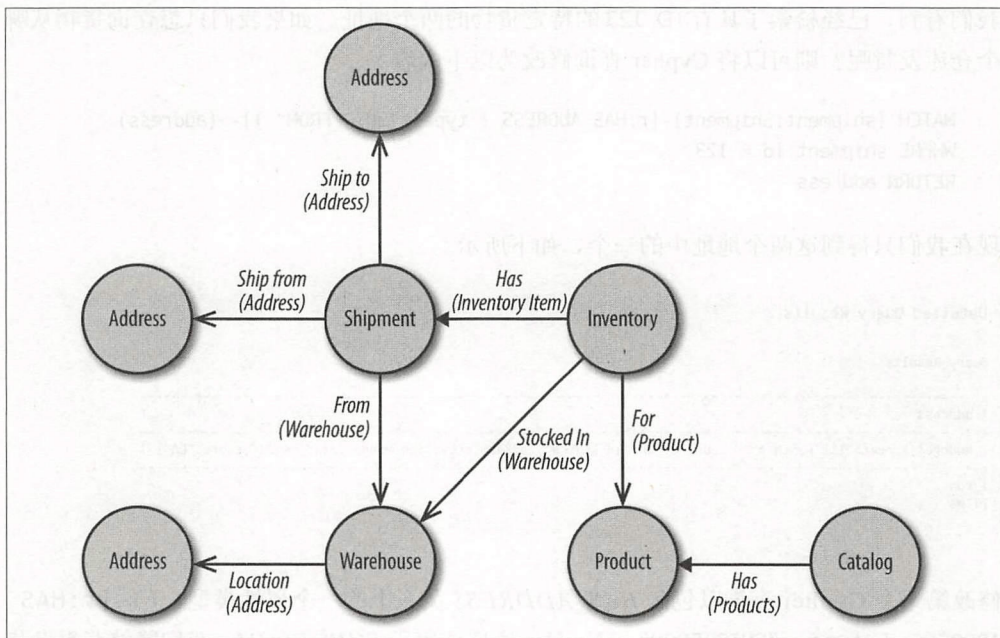


图9-13 库存有界上下文的图模型

我们需要在各自的包中为图 9-13 中描述的每个节点标签创建域类。第一个类是模型化 Address 的实体，如示例 9-26 所示。

示例9-26 Address域类作为Neo4j标签

```
package demo.address;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity
public class Address {

    @GraphId
    private Long id;

    private String street1, street2, state, city, country;
```



```

private Integer zipCode;

public Address(String street1, String street2, String state, String city,
    String country, Integer zipCode) {
    this.street1 = street1;
    this.street2 = street2;
    this.state = state;
    this.city = city;
    this.country = country;
    this.zipCode = zipCode;
}
}

```

如示例 9-27 所示，Product 作为 Neo4j 标签，它模型化可供销售的产品、其单价和产品 ID（你可能会在某些产品中看到它被称为 SKU）。

示例9-27 Product域类作为Neo4j标签

```

package demo.product;

import demo.catalog.Catalog;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

import java.util.HashSet;
import java.util.Set;

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity
public class Product {

    @GraphId
    private Long id;

    private String name, productId;

    private Double unitPrice;

    public Product(String name, String productId, Double unitPrice) {
        this.name = name;
    }
}

```

```

        this.productId = productId;
        this.unitPrice = unitPrice;
    }
}

```

Warehouse，如示例 9-28 所示，模型化了一个拥有物理货物结构的逻辑名称，以及该结构的位置。

示例9-28 Warehouse域类作为Neo4j标签

```

package demo.warehouse;

import demo.address.Address;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

@NoArgsConstructor
@AllArgsConstructor
@Data
@NodeEntity
public class Warehouse {

    @GraphId
    private Long id;

    private String name;

    @Relationship(type = "HAS_ADDRESS")
    private Address address;

    public Warehouse(String n, Address a) {
        this.name = n;
        this.address = a;
    }

    public Warehouse(String name) {
        this.name = name;
    }
}

```

Inventory 如示例 9-29 所示，Inventory 描述给定仓库有多少库存。

示例9-29 Inventory域类作为Neo4j标签

```
package demo.inventory;

import demo.product.Product;
import demo.warehouse.Warehouse;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity
public class Inventory {

    @GraphId
    private Long id;

    private String inventoryNumber;

    @Relationship(type = "PRODUCT_TYPE", direction = "OUTGOING")
    private Product product;

    @Relationship(type = "STOCKED_IN", direction = "OUTGOING")
    private Warehouse warehouse;

    private InventoryStatus status;

    public Inventory(String inventoryNumber, Product product, Warehouse warehouse,
        InventoryStatus status) {
        this.inventoryNumber = inventoryNumber;
        this.product = product;
        this.warehouse = warehouse;
        this.status = status;
    }
}
```

Shipment，如示例 9-30 所示，将一组库存中的信息模型化为一个出货 Address。

示例9-30 Shipment域类作为Neo4j标签

```
package demo.shipment;
```

```

import demo.address.Address;
import demo.inventory.Inventory;
import demo.warehouse.Warehouse;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

import java.util.HashSet;
import java.util.Set;

```

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity
public class Shipment {

    @GraphId
    private Long id;

    @Relationship(type = "CONTAINS_PRODUCT")
    private Set<Inventory> inventories = new HashSet<>();

    @Relationship(type = "SHIP_TO")
    private Address deliveryAddress;

    @Relationship(type = "SHIP_FROM")
    private Warehouse fromWarehouse;

    private ShipmentStatus shipmentStatus;

    public Shipment(Set<Inventory> inventories, Address deliveryAddress,
        Warehouse fromWarehouse, ShipmentStatus shipmentStatus) {
        this.inventories = inventories;
        this.deliveryAddress = deliveryAddress;
        this.fromWarehouse = fromWarehouse;
        this.shipmentStatus = shipmentStatus;
    }
}

```

Catalog, 如示例 9-31 所示, 是可用于某些 (可能是任意) 组名的可用产品的枚举。

示例9-31 Catalog域类作为Neo4j标签

```

package demo.catalog;

```



```

import demo.product.Product;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.neo4j.ogm.annotation.GraphId;
import org.neo4j.ogm.annotation.NodeEntity;
import org.neo4j.ogm.annotation.Relationship;

import java.util.Collection;
import java.util.HashSet;
import java.util.Set;
import java.util.stream.Collectors;

@Data
@NoArgsConstructor
@AllArgsConstructor
@NodeEntity
public class Catalog {

    @GraphId
    private Long id;

    @Relationship(type = "HAS_PRODUCT", direction = Relationship.OUTGOING)
    private Set<Product> products = new HashSet<>();

    private String name;

    public Catalog(String n, Collection<Product> p) {
        this.name = n;
        this.products.addAll(p);
    }

    public Catalog(String name) {
        this.name = name;
    }
}

```

Spring Data Neo4j 对本章前面的示例中使用的存储库使用相同的抽象。我们将在清单服务中再次使用 `PagingAndSortingRepository` 抽象，来指示 Spring Data 为存储库提供内置的分页和排序功能，如示例 9-32 所示。

示例9-32 对Neo4j中的地址标签分页和分类储存库

```

package demo.address;

import org.springframework.data.neo4j.repository.GraphRepository;

```

```
public interface AddressRepository extends GraphRepository<Address> {
}
```

在每个包含我们图模型中的域类的包中,我们将创建一个存储库,以便管理域数据。例如,示例 9-32 的片段是位于项目的地址包中的存储库定义。这个 `AddressRepository` 将允许我们使用标签 `Address` 来管理 Neo4j 中节点的数据。

现在我们的 `Inventory Service` 项目的包结构应该如图 9-14 所示。

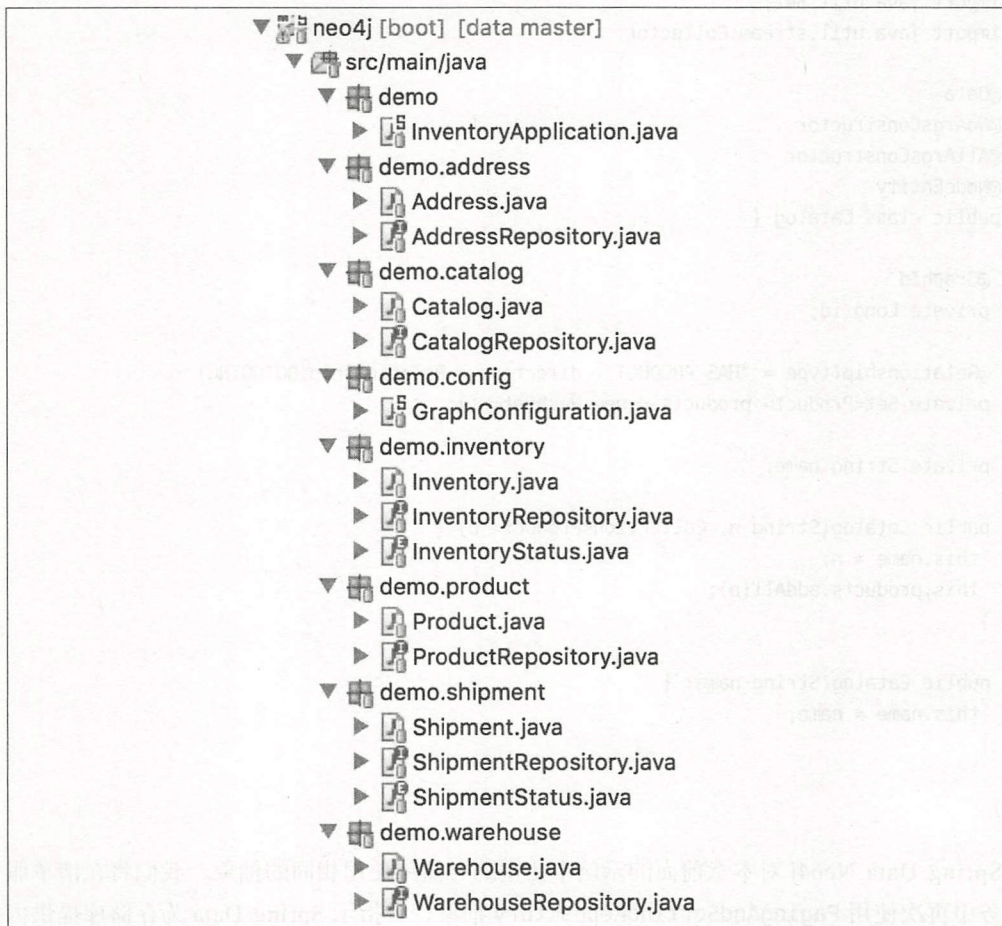


图9-14 清单服务的包结构

集成测试

我们已经为 `Inventory Service` 创建了数据层,现在再创建一个简单的端到端集成测试。

在测试中将采取以下步骤（如示例 9-33 所示）：

- 创建一个仓库。
- 创建一个产品列表。
- 创建一个目录。
- 创建发货地址。
- 为产品创建库存。
- 创建一批产品库存。

示例9-33 Neo4j图的集成测试

```
package demo;

import demo.address.Address;
import demo.address.AddressRepository;
import demo.catalog.Catalog;
import demo.catalog.CatalogRepository;
import demo.inventory.Inventory;
import demo.inventory.InventoryRepository;
import demo.product.Product;
import demo.product.ProductRepository;
import demo.shipment.Shipment;
import demo.shipment.ShipmentRepository;
import demo.shipment.ShipmentStatus;
import demo.warehouse.Warehouse;
import demo.warehouse.WarehouseRepository;
import org.apache.commons.lang3.exception.ExceptionUtils;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.neo4j.ogm.session.Session;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import java.util.*;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import static demo.inventory.InventoryStatus.*;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = InventoryApplication.class)
public class InventoryApplicationTests {
```

```

@Autowired
private ProductRepository products;

@Autowired
private ShipmentRepository shipments;

@Autowired
private WarehouseRepository warehouses;

@Autowired
private AddressRepository addresses;

@Autowired
private CatalogRepository catalogs;

@Autowired
private InventoryRepository inventories;

@Autowired
private Session session;

@Before
public void setup() {
    try {
        this.session.query("MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n, r;")
            .collections().emptyMap().queryResults();
    }
    catch (Exception e) {
        Assert.fail("can't connect to Neo4j! " + ExceptionUtils.getMessage(e));
    }
}

@Test
public void inventoryTest() {

    ❶
    List<Product> products = Stream
        .of(
            new Product("Best. Cloud. Ever. (T-Shirt, Men's Large)", "SKU-24642", 21.99),
            new Product("Like a BOSH (T-Shirt, Women's Medium)", "SKU-34563", 14.99),
            new Product("We're gonna need a bigger VM (T-Shirt, Women's Small)",
                "SKU-12464", 13.99),
            new Product("cf push awesome (Hoodie, Men's Medium)", "SKU-64233", 21.99))
        .map(p -> this.products.save(p)).collect(Collectors.toList());

    Product sample = products.get(0);
    Assert.assertEquals(this.products.findOne(sample.getId()).getUnitPrice(),

```



```
sample.getUnitPrice());
```

❷

```
this.catalogs.save(new Catalog("Spring Catalog", products));
```

❸

```
Address warehouseAddress = this.addresses.save(new Address("875 Howard St",
    null, "CA", "San Francisco", "United States", 94103));
Address shipToAddress = this.addresses.save(new Address(
    "1600 Amphitheatre Parkway", null, "CA", "Mountain View", "United States",
    94043));
```

❹

```
Warehouse warehouse = this.warehouses.save(new Warehouse("Pivotal SF",
    warehouseAddress));
Set<Inventory> inventories = products
    .stream()
    .map(
        p -> this.inventories.save(new Inventory(UUID.randomUUID().toString(), p,
            warehouse, IN_STOCK)).collect(Collectors.toSet());
Shipment shipment = shipments.save(new Shipment(inventories, shipToAddress,
    warehouse, ShipmentStatus.SHIPPED));
Assert.assertEquals(shipment.getInventories().size(), inventories.size());
}
}
```

❶ 保存一些产品。

❷ 创建一个目录。

❸ 创建一些发货地址。

❹ 创建一个货件。

Spring Data Redis

Spring Data Redis 项目为 Redis 键值存储提供了 Spring 集成能力。Redis 是一个开源的 NoSQL 数据库，也是当今使用最广泛的键值存储之一。虽然 Redis 被分类为键值存储，但最好将其解释为内存中的数据结构存储。Redis 与大多数其他数据存储不同，因为它能够将不同类型的复杂数据结构存储为值。Redis 最有趣的是，它为支持的每种数据结构提供了不同的操作集合。

Redis 支持的数据结构有：

- String
- List
- Set
- Hash
- Sorted set
- Bitmap 和 HyperLogLog

使用分布式数据存储（专为复杂数据结构的操作而设计）的好处是，多进程、应用程序和服务器能够使用相同的密钥对值进行并发操作。通常，要从多个应用程序执行此操作，需要在操作其值之前将某种形式的反序列化转换为语言支持的数据结构（例如，列表或集合）。

Redis 通过提供编程访问能力来解决这个问题，以便在其支持的数据结构上对 API 执行操作。这意味着对数据结构的操作可以被比序列化方式更大事务粒度原子地应用。

Redis 也经常用于进程间通信和消息传递。除了对数据结构操作提供专门支持之外，Redis 还可以作为消息代理，支持发布 / 订阅消息。

高速缓存

Redis 最流行的用例是缓存。Spring Data Redis 实现了 Spring 框架的 CacheManager 抽象，这使得它成为在微服务架构中集中缓存记录的绝佳选择。在本节中，我们将探索一个使用 *Spring Data Redis* 来管理用户记录缓存的 Spring Boot 应用程序。

正如我们在图 9-15 中所看到的，*User Service* 是 Spring Boot 应用程序，负责管理 *User* 域类的资源。我们也可以看到，有一个 *User Web* 应用程序。这个 Spring Boot 应用程序将成为 *User Service* 的使用者，并扮演一个依赖于 *User Service* 管理的 *User* 资源的 Web 应用程序的角色。

User Service 提供了一个 REST API，以允许消费者通过 HTTP 远程管理 *User* 资源。*User Web* 应用程序将承担托管单页应用的前端应用程序的角色，以管理用户。这个 Spring Boot 应用程序将托管静态 HTML/CSS 和 JavaScript，并使用从 *User Service* 公开的 REST API。



在图 9-15 中，蓝色表示无状态的服务，不依赖于附加的数据库。绿色是一种服务，它与数据库提供者有一个或多个连接，专门管理一组资源的持久性，在本案例下是 *User* 资源。

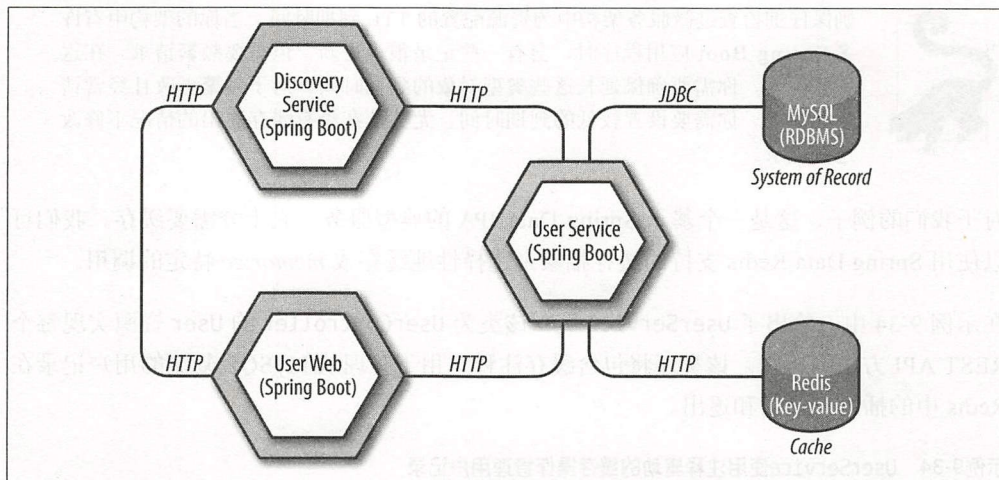


图9-15 *User Service*和*User Web*应用程序的服务架构图

当在 Spring Boot 应用程序中添加缓存层时，我们必须考虑数据使用的不同场景。缓存是任何微服务架构的重要组成部分，因为对于具有 REST API 的每个服务来说，与资源的交互需求非常大。我们可以从图 9-15 看到，有两个数据存储区：一个是 MySQL 数据库；另一个是 Redis 服务器。从 User Service 到其附加的 MySQL 数据库的记录将被存储并保存到磁盘上。当从另一个服务请求记录时，响应将记录复制到 Redis 服务器而被缓存，在 Redis 服务器中，记录保存在内存中，并且在再次缓存到期之前提供给消费者使用。

缓存的目标是通过使用辅助的高可用性存储来提高性能，并降低主数据的命中负载和连接池利用率。考虑 MySQL 数据库连接数量有限的场景。如果超过了数据库允许的连接数量，在连接被回收并释放回池之前，数据库可能会停机或不可用。为了解决这个问题，我们可以使用一个辅助存储器来存储内存中的记录，这样大部分流量将通过内存的缓存，从而节约了主数据库的连接和命中。这在微服务架构中是必不可少的，在微服务架构中，资源必须既能高效地服务于消费者，又能持久保存到磁盘上。

如果我们的类路径中包含 `org.springframework.boot:spring-boot-starter-data-redis`，Spring Boot 会为我们自动配置一个 `RedisTemplate`，在 `Environment` 中指定不同的配置值（`spring.redis.host`、`spring.redis.port` 等）。如果我们将 `@EnableCaching` 加入 `@Configuration` 类，它会更进一步地为我们配置一个 `Spring Cache CacheManager`。



确保仔细检查过微服务架构中为资源配置的 TTL 到期时间。当你的架构中有许多 Spring Boot 应用程序时，会有一些记录很少更新，但需要频繁请求。在这种情况下，你需要确保延长这些类型对象的到期时间。对于频繁更改且经常请求的资源，你需要设置较低的到期时间，尤其是在没有缓存逐出的情况下修改这些记录。

对于我们的例子，这是一个基于 Spring Data JPA 的典型服务，它十分需要缓存。我们可以使用 Spring Data Redis 支持的缓存抽象来选择性地缓存或 *memorize* 特定的调用。

在示例 9-34 中，给出了 `UserService` 类，该类为 `UserController` 的 `User` 资源实现每个 REST API 方法的逻辑。该类还将包含缓存注释，用于管理从 MySQL 复制的用户记录在 Redis 中的插入、检索和逐出。

示例9-34 `UserService`使用注释驱动的缓存操作管理用户记录

```
package demo;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;
```

```
@Service
```

```
public class UserService {
```

```
    private final UserRepository userRepository;
```

```
    @Autowired
```

```
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
```

```
    @CacheEvict(value = "user", key = "#user.getId()")
```

```
    ❶ public User createUser(User user) {
```

```
        User result = null;
```

```
        if (!userRepository.exists(user.getId())) {
            result = this.userRepository.save(user);
        }
```

```
        return result;
    }
```



```
@Cacheable(value = "user")
```

②

```
public User getUser(String id) {  
    return this.userRepository.findOne(id);  
}
```

```
@CachePut(value = "user", key = "#id")
```

③

```
public User updateUser(String id, User user) {  
  
    User result = null;  
  
    if (userRepository.exists(user.getId())) {  
        result = this.userRepository.save(user);  
    }  
  
    return result;  
}
```

```
@CacheEvict(value = "user", key = "#id")
```

④

```
public boolean deleteUser(String id) {  
  
    boolean deleted = false;  
  
    if (userRepository.exists(id)) {  
        this.userRepository.delete(id);  
        deleted = true;  
    }  
  
    return deleted;  
}  
}
```

- ❶ 使用 Redis 缓存中提供的 ID 来逐出 User 记录。
- ❷ 从 Redis 获取缓存记录，或者将 MySQL 的 User 记录放入 Redis 缓存中。
- ❸ 清除 User 记录，并用新更新的 User 记录替换它。
- ❹ 使用 Redis 缓存中提供的 ID 清除 User 记录。

在示例 9-34 中，UserService 类负责执行缓存操作以及管理 MySQL 中 User 实体类的数据库记录。此服务使用注释驱动的缓存来管理 Redis 中复制记录的生命周期。本例中的 4 个 CRUD 操作相当简单。User 记录的主键被用来为 Redis 提供的缓存管理器生成密钥。缓存注释使用 Spring 表达式语言（SpEL）从方法的参数访问密钥。



将 Redis 服务器共享为不同服务的共享缓存时，务必记住生成的密钥中不包含服务名称的命名空间。这意味着如果对于不同服务管理的资源的不同域名概念使用了相同的标识符，将会产生冲突，导致一个敏感资源被另一个敏感资源替代。在这种情况下，当请求 User 记录时，你可能会得到 Account 记录。共享缓存时一定要小心。确保为拥有缓存记录的服务生成使用唯一命名空间的密钥。

Redis 可以作为任意数量的服务的缓存骨干。在这个例子中，我们研究了缓存，但是你也可以使用 Redis 作为 HTTP 会话的后端，这个话题我们在第 5 章中进行了探讨，即通过 Spring Session 将传统应用转移到云环境。

总结

在本章中，我们探讨了如何创建基于各种 Spring Data 项目的 Spring Boot 应用程序。我们为 *Cloud Native Clothing* 的在线商店创建了一个完整的后端，在每个应用程序中使用了不同的 Spring Data 项目：

- Account Service (JPA-MySQL)
- Order Service (MongoDB)
- Inventory Service (Neo4j)

我们还研究了如何使用 Spring Data Redis 来缓存基于这些数据源的服务。

消息系统

通过消息系统可以跨越进程和网络来连接应用服务公开的事件。消息系统的应用有很多，Martin Fowler 的这篇博客涵盖了很多这种应用：*What do you mean by “Event-Driven”?* (<https://martinfowler.com/articles/201701-event-driven.html>)。事件驱动的含义如下。

- **事件通知**：通过发送事件消息来通知其他系统其系统域中的变化。消息的接收者不产生答复。源系统不期望应答，也不需要应答。事件通知是不可变的，这意味着事件消息的内容不应包含事件生成后修改的数据。
- **携事件的状态转移**：消息中不包含任何要求接收方回调源系统的数据。各式各样的事件消息中包括接收者处理事件所需的一切内容。
- **事件溯源**：事件溯源是对存储能够导致系统状态随时间变化的领域事件日志的实践。在这种情况下，可以从任何时间点重新播放事件以重建系统状态。

消息代理（如 Apache Kafka、RabbitMQ、ActiveMQ 和 MQSeries）充当消息的存储库。生产者和消费者连接到消息代理，而不是彼此直接连接。传统的代理是静态的，但是像 RabbitMQ 和 Apache Kafka 这样的产品可以根据需要进行扩展。消息系统将生产者和消费者分离开来。生产者和消费者不需要在相同的位置，存在于相同的地点或进程中，甚至可以不同时可用。在云环境中，这些特性更加重要，因为云环境中的服务是流动且短暂的。云平台支持弹性，可以根据需求增长和缩减。消息系统是在系统停机时抑制负载，在需要扩大规模时进行负载均衡的理想方式。



像 RabbitMQ 和 Apache Kafka 这样的技术很重要，但是它们会增加运营成本。理想情况下，它们应该由平台来管理，并且是自动化的。如果你使用的是 Cloud Foundry，则已经包含 RabbitMQ 服务。

Spring Integration 的事件驱动架构

我们希望一切都能够按照我们的计划有条不紊地运行，但是真实世界并非如此：周围的事件驱动着我们所做的一切。数据的逻辑窗口是有价值的，但有些数据不能用窗口来处理。有些数据是连续的并且与现实世界中的事件相关联。在本节中，我们将介绍如何使用由事件驱动的数据。

当我们讨论事件时，大多数人可能会想到消息传递技术，比如 JMS、RabbitMQ、Apache Kafka、Tibco Rendezvous 或 IBM MQSeries。通过将消息发送到集中式中间件来连接不同的自治客户端，就像电子邮件让人们互相连接一样。消息代理存储和传递消息，直到客户端可以消费和响应（与电子邮件收件箱完全相同）。

大多数消息传递技术都提供 API 和客户端。Spring 一直以来都对消息传递技术有很好的底层支持，你可以找到对 JMS API、AMQP（以及像 RabbitMQ 这样的代理）、Redis 和 Apache Geode 的复杂的底层支持。

当然，世界上有很多种事件类型。接收电子邮件就是一个例子。接收推文又是一个例子。在目录中创建了新文件这这也是一个事件。一个 XMPP 驱动的聊天消息也是一个事件。MQTT 微波发送状态更新也是一个事件。

这实在是太复杂了！如果你查看不同事件源的生态图谱，那么你会（希望）看到很多选择以及它的复杂性。有些复杂性来自整合本身。如何建立一个依赖于这些不同系统事件的系统？你可能会根据各种事件源之间的点对点连接来解决集成问题，但这最终会导致 *spaghetti architecture*（意大利面条架构）。在数学上来看这也不是个好想法，因为每个集成点都需要与其他所有集成点连接。这是一个二项式系数： $n(n-1)/2$ 。因此，如果有 6 个服务，就需要 15 个不同的点对点连接！

相反，我们采取更加结构化的方式来整合 Spring Integration。Spring Integration 的核心是 Spring 框架中的 `MessageChannel` 和 `Message <T>` 类型。`Message <T>` 对象有一个有效载荷（payload）和一组 header，用来提供有关消息有效载荷的元数据。`MessageChannel` 就像 `java.util.Queue` 一样。`Message <T>` 对象流过 `MessageChannel` 实例。

Spring Integration 可以跨多个不兼容的系统集成服务和数据。从概念上讲，组合一个集成流程类似于使用 `stdin` 和 `stdout` 在 UNIX 操作系统上编写管道和过滤器流程（如示例 10-1 所示）。

示例10-1 使用管道和过滤器模型连接每个命令行的实用程序

```
cat input.txt | grep ERROR | wc -l > output.txt
```


在这里,我们从一个源文件(文件 `input.txt`)获取数据,将其传递给 `grep` 命令来过滤结果,只保留包含 `ERROR` 的行。然后,将其传递到 `wc` 实用程序来计算有多少行。最后计数被写入输出文件 `output.txt`。组件 `cat`、`grep` 和 `wc` 彼此互不知道对方的存在。在设计它们的时候就没有考虑过这个问题。相反,它们只知道如何从 `stdin` 读取数据并写入 `stdout`。这种数据的规范化使得由简单的原子组成复杂的解决方案变得非常容易。在这个例子中,`cat` 命令将文件数据转换为任何了解 `stdin` 的进程都可以读取的数据。它将入站数据转换为标准化格式 `stdout` 上的字符串行。最后,重定向 (`>`) 操作符将归一化数据(字符串行)转换为文件系统中的数据。管道 (`|`) 字符用于表示一个组件的输出应该流向另一个组件的输入。

Spring Integration 流程的工作方式与此相同:数据被标准化为 `Message <T>` 实例。每个 `Message <T>` 实例都有一个 `payload` 和 `header` 信息,其中有 `Map <K,V>` 格式的 `payload` 元数据,即不同消息传递组件的输入和输出。这些消息传递组件通常由 Spring Integration 提供,但编写和使用自己的代码也很容易。有各种消息组件来支持所有企业应用程序集成模式 (<http://www.enterpriseintegrationpatterns.com/>) (过滤器、路由器、变换器、适配器、网关等)。Spring 框架中的 `MessageChannel` 是一个命名管道,`Message <T>` 在消息组件之间流动。管道默认的工作方式类似于 `java.util.Queue`。数据输入其中后输出。

消息端点

通过消息传递端点连接 `MessageChannel` 对象:不同的 Java 对象处理不同类型的消息。当你传递一个 `Message <T>`,或者只是传递一个 `T` 给各个组件时, Spring Integration 就可以正常工作。Spring Integration 提供了组件模型和 Java DSL。Spring Integration 流中的每个消息传递终端都可能产生一个输出值,然后将其发送给下游或者是 `null` 来终止处理。

入站网关 (*inbound gateway*) 接收来自外部系统的传入请求,作为 `Message <T>` 处理,并发送回复。出站网关 (*outbound gateway*) 采用 `Message <T>`,将它们转发到外部系统,并等待来自该系统的响应。它们支持请求和回复交互。

入站适配器 (*inbound adapter*) 是从外部接收消息并将它们变成 Spring `Message <T>` 的组件。出站适配器 (*outbound adapter*) 做相同的事情,只不过它接收 Spring `Message <T>` 并将其转换为下游系统期望的消息类型传递出去。Spring Integration 包含一系列使用不同技术和协议的适配器,包括 MQTT、Twitter、电子邮件、(S) FTP (S)、XMPP、TCP/UDP 等。

入站适配器有两种:轮询适配器 (*polling adapter*) 和事件驱动的适配器 (*event-driven adapter*)。入站轮询适配器以一定的时间间隔或以由上游消息源定义的速率自动启动。

网关 (*gateway*) 是一个用来处理请求和回复的组件。例如, 入站网关从外部接收一条消息, 将其传递给 Spring Integration, 然后将回复消息发给外部世界; 典型的 HTTP 流就是这样的。出站网关采用 Spring Integration 消息并将其传递到外部世界, 然后在 Spring Integration 中将其回传。如果你指定了回复目标, 则在使用 RabbitMQ 代理时可能会看到。

过滤器 (*filter*) 是一个应用某种条件来确定是否应该继续传入消息的组件。可以把它想像成 Java 中的 `if (...)` 测试。

路由器 (*router*) 接收传入的消息并应用一个测试 (任何你想要的测试) 来决定下一个消息的发送位置。可以将路由器看作消息的交换语句。

转换器 (*transformer*) 接收消息, 可以对消息做任何事情, 例如扩展或者改变消息, 然后将消息发送出去。

分解器 (*splitter*) 接收一个消息, 然后根据消息的某些属性将它分成多个较小的消息, 然后再向下游转发。例如, 你可能会收到订单传入消息, 然后按顺序将每个订单项的消息转发给某种履行流程。

聚合器 (*aggregator*) 通过一些独特的属性关联, 将多条消息合并成一个消息, 然后发送到下游。

集成流程需要感知系统, 但是集成中使用的相关组件却不需要。这使得以小型或孤立的服务组成复杂的解决方案变得更加容易。构建 Spring Integration 流程的行为本身就是有益的。因为这样可以迫使你将服务的逻辑分解, 所有服务必须能够通过包含有效载荷的消息通信。有效载荷的模式 (*schema*) 是合约。该属性在分布式系统中非常有用。

使用简单的组件构建复杂的系统

Spring Integration 支持事件驱动架构, 因为它可以检测和响应外部事件。例如, 你可以使用 Spring Integration 每 10s 查询一次文件系统, 并在出现新文件时发布一个 `Message<T>`。你可以将 Spring Integration 作为 Apache Kafka topic 消息侦听器。适配器处理对外部事件的响应, 这样你就不必担心发起过多的消息, 并在消息到达时专注于处理消息。这种集成等价于依赖注入。

依赖注入 (*dependency injection*) 使组件代码不用关心资源的初始化和获取, 其可以自由地专注于编写这些依赖关系的代码。`javax.sql.DataSource` 字段来自哪里? 谁在乎! Spring 将它连接起来。它可能来自测试中的模拟、应用程序服务器中的 JNDI 或者 Spring Boot bean 配置。组件代码仍然不知道这些细节。我们可以用“好莱坞原则”解释依赖注入: “不要打电话给我, 我会打给你的!”

被依赖的对象会被提供一个对象，而不是针对对象做初始化或资源查找。同样的原则也适用于 Spring Integration：编写代码的时候不知道消息来自何处。这大大简化了开发。

那么让我们来看一个简单的例子吧，在这个例子中对目录中出现新文件作出响应，记录观察到的文件，然后使用路由器进行简单的测试后将负载动态路由到两个可能的流之一。

我们将使用 Spring Integration Java DSL，它非常适合于 Java 8 中的 lambda。每个 IntegrationFlow 都隐式地将组件链接在一起。通过提供连接的 MessageChannel 引用来显式链接，如示例 10-2 所示。

示例10-2 使用管道和过滤器模型来连接独立的命令行实用程序

```
package eda;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.channel.MessageChannels;
import org.springframework.integration.dsl.file.Files;
import org.springframework.messaging.MessageChannel;

import java.io.File;

@Configuration
public class IntegrationConfiguration {

    private final Log log = LogFactory.getLog(getClass());

    @Bean
    IntegrationFlow etlFlow(
        @Value("${input-directory:${HOME}/Desktop/in}") File dir) {

        return IntegrationFlows
            ❶ .from(Files.inboundAdapter(dir).autoCreateDirectory(true),
                consumer -> consumer.poller(spec -> spec.fixedRate(1000)))
            ❷ .handle(File.class, (file, headers) -> {
                log.info("we noticed a new file, " + file);
                return file;
            })
            ❸
```

```

        .routeToRecipients(
            spec -> spec.recipient(csv(), msg -> hasExt(msg.getPayload(), ".csv"))
                .recipient(txt(), msg -> hasExt(msg.getPayload(), ".txt")))
        .get();
}

```

```

private boolean hasExt(Object f, String ext) {
    File file = File.class.cast(f);
    return file.getName().toLowerCase().endsWith(ext.toLowerCase());
}

```

④

```

@Bean
MessageChannel txt() {
    return MessageChannels.direct().get();
}

```

⑤

```

@Bean
MessageChannel csv() {
    return MessageChannels.direct().get();
}

```

⑥

```

@Bean
IntegrationFlow txtFlow() {
    return IntegrationFlows.from(txt()).handle(File.class, (f, h) -> {
        log.info("file is .txt!");
        return null;
    }).get();
}

```

⑦

```

@Bean
IntegrationFlow csvFlow() {
    return IntegrationFlows.from(csv()).handle(File.class, (f, h) -> {
        log.info("file is .csv!");
        return null;
    }).get();
}
}

```

- ❶ 配置一个 Spring Integration 入站适配器 File，告诉它如何消费传入的消息，以及轮询器应该以多少毫秒的速率扫描目录。
- ❷ 该方法宣布已经收到一个文件，然后转发有效载荷。

- ③ 通过众所周知的 `MessageChannel` 实例，将请求路由到两个可能的集成流之一，这两个集成流来源于文件的扩展。
- ④ 扩展名为 `.txt` 的文件将通过该通道传输。
- ⑤ 扩展名为 `.csv` 的文件将通过该通道传输。
- ⑥ `IntegrationFlow` 处理 `.txt` 文件。
- ⑦ `IntegrationFlow` 处理 `.csv` 文件。

通道 (channel) 是对逻辑的解耦，只要有一个指向它的通道，通道两端是什么都无所谓。通道的消费者可能今天是一个简单的日志记录 `MessageHandler <T>`，就像这个例子，但是明天又可能是一个向 Apache Kafka 写消息的组件。我们可以在消息到达通道的那一刻启动一个流程。至于数据到底是如何到达通道的就无关紧要了。我们可以接收来自 REST API 的请求，或者适配来自 Apache Kafka 的消息，或者监视一个目录。只要能够以某种方式将传入的消息调整为 `java.io.File` 并将其提交给正确的通道就可以了。

假设我们有一个处理文件的批处理流程。传统上，这样的工作可能会在固定的时间运行，也可能使用像 `cron` 这样的调度器。这样在两次运行之间会有空闲时间，而空闲时间延迟了我们期待的结果的出现。相反，只要出现一个新的 `java.io.File`，我们就可以使用 Spring Integration 启动一个批处理作业。Spring Integration 提供了一个入站文件适配器。我们将监听来自入站文件适配器的消息，然后将它们转换为有效负载为 `JobLaunchRequest` 的消息。`JobLaunchRequest` 描述了要启动的 `Job`，并描述了该作业的 `JobParameters`。最后，`JobLaunchRequest` 被转发到 `JobLaunchingGateway`，然后生成一个 `JobExecution` 对象，我们检查它并决定在哪里执行路由。如果作业正常完成，我们把输入文件移到已完成作业的目录中。否则，将文件移动到错误目录中。

将有一个主流程将执行转发到两个执行分支之一，它们由两个通道表示：`invalid` 和 `completed` (如示例 10-3 所示)。

示例10-3 两个MessageChannel实例

```
package edabatch;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.channel.MessageChannels;
import org.springframework.messaging.MessageChannel;

@Configuration
class BatchChannels {
```

```

@Bean
MessageChannel invalid() {
    return MessageChannels.direct().get();
}

@Bean
MessageChannel completed() {
    return MessageChannels.direct().get();
}
}

```

主流程（称为 `etlFlow`）以固定的速率监视一个目录（`directory`），并将每个事件转换成 `JobLaunchRequest`（如示例 10-4 所示）。

示例10-4 两个 `EtlFlowConfiguration` 实例

```

package edabatch;

import org.springframework.batch.core.*;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.batch.integration.launch.JobLaunchRequest;
import org.springframework.batch.integration.launch.JobLaunchingGateway;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.file.Files;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;

import java.io.File;

import static org.springframework.integration.file.FileHeaders.ORIGINAL_FILE;

@Configuration
class EtlFlowConfiguration {

    ❶
    @Bean
    IntegrationFlow etlFlow(
        @Value("${input-directory:${HOME}/Desktop/in}") File directory,
        BatchChannels c, JobLauncher launcher, Job job) {

        return IntegrationFlows

```



```

.from(Files.inboundAdapter(directory).autoCreateDirectory(true),
cs -> cs.poller(p -> p.fixedRate(1000)))
.handle(
File.class,
(file, headers) -> {

String absolutePath = file.getAbsolutePath();

②
JobParameters params = new JobParametersBuilder().addString("file",
absolutePath).toJobParameters();

return MessageBuilder.withPayload(new JobLaunchRequest(job, params))
.setHeader(ORIGINAL_FILE, absolutePath)
.copyHeadersIfAbsent(headers).build();
})

③
.handle(new JobLaunchingGateway(launcher))

④
.routeToRecipients(
spec -> spec.recipient(c.invalid(), this::notFinished).recipient(
c.completed(), this::finished)).get();
}

private boolean finished(Message<?> msg) {
Object payload = msg.getPayload();
return JobExecution.class.cast(payload).getExitStatus()
.equals(ExitStatus.COMPLETED);
}

private boolean notFinished(Message<?> msg) {
return !this.finished(msg);
}
}

```

- ① 这个 `EtlFlowConfiguration` 和前面的例子中的一样。
- ② 使用 `JobParametersBuilder` 为 Spring Batch 作业设置 `JobParameters`。
- ③ 将作业和相关参数转发到 `JobLaunchingGateway`。
- ④ 通过检查 `JobExecution` 来测试作业是否正常退出。

Spring Integration 流程中的最后一个组件是路由器，路由器检查来自 `Job` 的 `ExitStatus`，并确定输入文件应该被移动到哪个目录。如果文件成功被完成，它将被路由到 `completed` 通道。如果出现错误或者由于某种原因而提前终止作业，它将被路由到 `invalid` 通道。

FinishedFileFlowConfiguration 配置监听 completed 通道上传入的消息，将传入的有效载荷（java.io.File）移动到 completed 目录，然后查询写入数据的表（如示例 10-5 所示）。

示例10-5 FinishedFileFlowConfiguration实例

```
package edabatch;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.batch.core.JobExecution;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.file.FileHeaders;
import org.springframework.jdbc.core.JdbcTemplate;

import java.io.File;
import java.util.List;

import static edabatch.Utils.mv;

@Configuration
class FinishedFileFlowConfiguration {

    private Log log = LogFactory.getLog(getClass());

    @Bean
    IntegrationFlow finishedJobsFlow(BatchChannels channels,
        @Value("${completed-directory:${HOME}/Desktop/completed}") File finished,
        JdbcTemplate jdbcTemplate) {
        return IntegrationFlows
            .from(channels.completed())
            .handle(JobExecution.class,
                (je, headers) -> {
                    String ogFileName = String.class.cast(headers
                        .get(FileHeaders.ORIGINAL_FILE));
                    File file = new File(ogFileName);
                    mv(file, finished);
                    List<Contact> contacts = jdbcTemplate.query(
                        "select * from CONTACT",
                        (rs, i) -> new Contact(
                            rs.getBoolean("valid_email"),
                            rs.getString("full_name"),
                            rs.getString("email"),
```



```

        rs.getLong("id")));
        contacts.forEach(log::info);
        return null;
    }).get();
}
}

```

InvalidFileFlowConfiguration 配置监听 invalid 通道上传入的消息，并将传入的有效载荷（java.io.File）移动到 errors 目录，然后终止流程（如示例 10-6 所示）。

示例10-6 InvalidFileFlowConfiguration实例

```

package edabatch;

import org.springframework.batch.core.JobExecution;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.file.FileHeaders;

import java.io.File;

import static edabatch.Utils.mv;

@Configuration
class InvalidFileFlowConfiguration {

    @Bean
    IntegrationFlow invalidFileFlow(BatchChannels channels,
        @Value("${error-directory:${HOME}/Desktop/errors}") File errors) {
        return IntegrationFlows
            .from(channels.invalid())
            .handle(JobExecution.class,
                (je, headers) -> {
                    String ogFileName = String.class.cast(headers
                        .get(FileHeaders.ORIGINAL_FILE));
                    File file = new File(ogFileName);
                    mv(file, errors);
                    return null;
                }).get();
    }
}

```

MessageChannel 定义用于分离不同的集成流。我们可以重复使用通用功能，并构建仅使

用通道作为连接的高阶系统。MessageChannel 是接口。

消息代理、桥接、竞争消费者模式和事件溯源

最后一个例子非常简单。就消息（事件）而言这是一个简单的流程。我们能够尽快开始处理消息而不会使系统过载。由于集成是根据通道构建的，每个组件都可以使用通道消费或生成消息，因此在入站文件适配器和实际启动 Spring Batch 作业的节点之间引入消息代理将十分烦琐。但是为了扩展集群中的工作，可以由 Cloud Foundry 这样的平台提供支持。

也就是说，你不太可能将文件系统集成到云原生架构中。但是，你可能有其他非事务性事件源（消息生产者）和接收器（消息消费者）需要与其集成。

我们可以使用 saga 模式和设计补偿事务来处理我们集成的每个服务以及任何可能的失败情况，但是如果使用消息代理，事情可能会更简单一些。消息代理在概念上非常简单：当消息被传送给代理时，它们被存储并传递给连接的消费者。如果没有连接的消费者，代理将存储消息，并在消费者连接时重新发送消息。

消息代理通常提供两种类型的目的地（可以将它们视为邮箱）：发布—订阅和点对点。

发布—订阅目的地

发布—订阅目的地向所有连接的消费者发送一条消息。这有点像通过扩音器向整个房间传播信息。

发布—订阅消息传递支持事件协作，其中多个系统保持自己的状态。当新事件从系统中的不同组件到达时，随着域对象状态的更新，每个系统将通过更新其应用程序状态的视图来作出反应。

假设你有一个产品目录。当向 *ProductService* 中添加新条目时，它可能会发布描述增量的事件。*SearchEngine* 服务可能会消费消息，然后更新本地绑定的 *ElasticSearch* 服务实例上的索引。*InventoryService* 可能会更新包含在本地绑定的 *RDBMS* 服务实例中的数据。*RecommendationService* 可能决定更新本地绑定的 *Neo4j* 服务实例中的数据。这些系统不再需要询问 *ProductService*。

如果在有序日志中记录每个域事件，则可以通过在任何以前的时间点重新创建系统状态来执行临时查询。如果有任何服务失败，则其本地状态可以从日志中完全被重新创建。这种做法被称为事件溯源，这是建立分布式系统非常强大的工具。

点对点目的地

即使存在多个连接的消费者，点对点目的地也会向其中的每个消费者发送一条消息。这有点像秘密告诉小组中的某一个人。如果连接了多个消费者，并且他们都尽可能快地从点对点的目的地消费信息，这有些负载均衡的意味：工作量被连接的消费者数量分割。这种被称为竞争性消费者模式的方法简化了多个消费者之间的负载均衡工作。这是利用 Cloud Foundry 等云计算环境的弹性的理想方式，其中水平容量是弹性的和（实际上）无限的。

消息代理也有自己的资源本地化的事务概念。生产者可能会在发送一条消息后在必要的时候将其撤回，从而有效地回滚消息。消费者可以接受消息的传递，尝试对消息进行处理，然后确认交付；或者如果出现问题，则将消息返回给代理，从而有效地回滚交付。最终双方会就状态达成一致。这与分布式事务的不同之处在于，消息代理引入了时间变量或时间解耦。这样做简化了服务之间的集成。这个属性使得在分布式系统中更容易推理状态。这样可以确保两个非事务性的资源在最终状态上达成一致。通过这种方式，消息代理桥接了另外两个非事务性资源。

消息代理向系统中添加了可移动部分使得架构更加复杂。消息代理需要拥有众所周知的灾难恢复、备份和扩展方案。每个组织都应该知道如何做到这一点，这可以在所有服务中重复使用。另一种选择是每个服务都被迫重新创建，这要么不那么理想，要么缺少那些特性。如果你正在使用 Cloud Foundry 这样的平台，而 Cloud Foundry 已经为你管理了消息代理，那么使用消息代理应该是一件非常容易的事情。

从逻辑上讲，消息代理作为连接不同服务的方式是很有意义的。Spring Integration 对此提供了足够的支持，使用适配器来生成和消费不同代理的消息。

Spring Cloud Stream

虽然 Spring Integration 在解决与消息代理的服务间通信问题方面有着坚实的基础，但对于微服务来说，它似乎有些笨拙。我们希望就像能够使用 Spring 进行基于 REST 的交互一样，Spring 也能够支持消息传递。我们不会使用 Twitter 或电子邮件连接服务，而更倾向于使用具有已知交互模式的 RabbitMQ、Apache Kafka 或类似的消息代理。当然，我们可以明确地配置入站和出站 RabbitMQ 或 Apache Kafka 适配器。相反，我们将视线移向技术堆栈的上层以简化工作，并消除我们每次想使用其他服务时就配置入站和出站适配器的主观意识。

Spring Integration 在解析 MessageChannel 对象方面做了很多工作。MessageChannel 是一种很好的间接寻址方式。从应用程序逻辑的角度来看，channel 代表了通过消息代理路

由下游服务的逻辑通道。

本节我们来了解一下 Spring Cloud Stream。Spring Cloud Stream 位于 Spring Integration 之上，通道是交互模型的核心。这意味着约定支持配置的简单外化。有了它，与消息代理连接的服务就变得更加简洁和清晰。

我们来看一个简单的例子。首先建立一个生产者和一个消费者。生产者将公开一个 REST API 端点，当其被调用时，该端点将消息发布到两个通道：一个用于 broadcast，或者说发布一订阅式消息传递；另一个用于点对点消息传递。然后，启动一个消费者接收这些消息。

通过 Spring Cloud Stream，可以轻松定义随后连接到消息系统上的通道。我们可以按照惯例使用 *binder* 实现——连接到代理。在这个例子中，我们将使用 RabbitMQ，这是一款流行的基于 AMQP 规范的消息代理。Spring Cloud Stream 的 RabbitMQ 支持的 binder 是 `org.springframework.cloud:spring-cloud-starter-stream-rabbit`。

因为有许多语言的客户端和绑定，这使得 RabbitMQ（和一般的 AMQP 规范）非常适合对跨不同语言和平台进行集成。Spring Cloud Stream 基于 Spring Integration（它提供了必要の入站和出站适配器），而 Spring Integration 依次构建 Spring AMQP（它提供了低级别的 `AmqpOperations`、`RabbitTemplate`、`RabbitMQ`、`ConnectionFactory` 等）。Spring Boot 根据默认值或属性自动配置一个 `ConnectionFactory`。在一个没有任何 RabbitMQ 实例的本地机器上，这个应用程序可以开箱即用。

流生产者

Spring Cloud Stream 的核心是绑定（*binding*）。绑定通过 `MessageChannel` 实例定义对其他服务的逻辑引用，并将把这些实例留给 Spring Cloud Stream 来连接。从业务逻辑的角度来看，这些下游或上游消息传递服务在 `MessageChannel` 对象的另一侧是未知的。目前，我们不需要担心如何建立连接。我们定义两个通道：一个通道向所有消费者广播一条问候语，另一个通道将发送一条点对点问候，看看哪个消费者首先接收到消息（如示例 10-7 所示）。

示例10-7 一个简单的以 `MessageChannel` 为中心的问候生产者

```
package stream.producer;
```

```
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;
```

```
public interface ProducerChannels {
```



```

❶
String DIRECT = "directGreetings";

String BROADCAST = "broadcastGreetings";

@Output(DIRECT)
❷
MessageChannel directGreetings();

@Output(BROADCAST)
MessageChannel broadcastGreetings();
}

```

- ❶ 默认情况下，流的名称（我们将在系统的其他部分使用）基于 MessageChannel 方法本身。在接口中提供一个字符串常量是很有用的，这样我们就可以直接引用。
- ❷ Spring Cloud Stream 提供了两个注解：@Output 和 @Input。@Output 注解告诉 Spring Cloud Stream 放入通道的消息将被发送出去（通常通过 Spring Integration 中的出站通道适配器发送）。

我们需要告诉 Spring Cloud Stream 如何处理发送到这些通道的数据，可以利用环境属性来实现。示例 10-8 显示了生产者的 application.properties 配置。

示例10-8 一个简单的以MessageChannel为中心的问候生产者

```

spring.cloud.stream.bindings.broadcastGreetings.destination=greetings-pub-sub ❶
spring.cloud.stream.bindings.directGreetings.destination=greetings-p2p

spring.rabbitmq.addresses=localhost ❷

```

- ❶ 在这两行中，spring.cloud.stream.bindings. 和 .destination 之间的部分必须匹配 Java MessageChannel 的名字。这是应用程序在其调用的服务上的本地视角。= 符号后面的位是生产者和消费者约定的会合点。双方都需要在这里指定确切的名字。这是我们配置的代理目的地的名称。
- ❷ 我们使用 Spring Boot 的自动配置来创建一个 RabbitMQ ConnectionFactory，Spring Cloud Stream 将依赖它。

在示例 10-9 中，我们将看到一个简单的使用 REST API 的生产者，然后其向消费者发布消息。

示例10-9 一个简单的以MessageChannel为中心的问候生产者

```

package stream.producer.channels;

```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.http.ResponseEntity;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import stream.producer.ProducerChannels;

@SpringBootApplication
@EnableBinding(ProducerChannels.class)
1 public class StreamProducer {

    public static void main(String args[]) {
        SpringApplication.run(StreamProducer.class, args);
    }
}

@RestController
class GreetingProducer {

    private final MessageChannel broadcast, direct;

    2 @Autowired
    GreetingProducer(ProducerChannels channels) {
        this.broadcast = channels.broadcastGreetings();
        this.direct = channels.directGreetings();
    }

    @RequestMapping("/hi/{name}")
    ResponseEntity<String> hi(@PathVariable String name) {
        String message = "Hello, " + name + "!";

    3 this.direct.send(MessageBuilder.withPayload("Direct: " + message).build());

        this.broadcast.send(MessageBuilder.withPayload("Broadcast: " + message)
            .build());
        return ResponseEntity.ok(message);
    }
}

```


- ❶ `@EnableBinding` 注解启用 Spring Cloud Stream。
- ❷ 注入 `ProducerChannels`，然后在构造函数中解引用所需的通道，这样当有人向 `/hi/{name}` 发出 HTTP 请求时就可以发送消息。
- ❸ 这是一个普通的 Spring 框架通道，所以使用 `MessageBuilder` API 创建 `Message <T>` 足够简单。

当然，风格也很重要，尽管我们已经减少了使用到接口的下行服务，通过几行属性声明和几行以消息传递为中心的通道操作减少工作成本，但是如果使用 Spring Integration 的消息传递网关，效果将会更好。消息传递网关作为一种设计模式，它将客户端从服务背后的消息传递逻辑中隐藏起来。从客户端的角度来看，网关看起来可能像一个普通的对象。这样非常方便。你可以先定义一个接口和同步服务，然后将它作为基于消息传递网关的实现，在将来提取出来。我们重温一下使用生产者而不是直接使用 Spring Integration 发送消息，现在我们将使用消息传递网关发送消息（如示例 10-10 所示）。

示例10-10 消息传递网关生产者实现

```
package stream.producer.gateway;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.http.ResponseEntity;
import org.springframework.integration.annotation.Gateway;
import org.springframework.integration.annotation.IntegrationComponentScan;
import org.springframework.integration.annotation.MessagingGateway;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import stream.producer.ProducerChannels;
```

```
@SpringBootApplication
```

```
@EnableBinding(ProducerChannels.class)
```

```
❶
```

```
@IntegrationComponentScan
```

```
❷
```

```
public class StreamProducer {
```

```
    public static void main(String args[]) {
        SpringApplication.run(StreamProducer.class, args);
    }
```

```

}

③
@MessagingGateway
interface GreetingGateway {

    @Gateway(requestChannel = ProducerChannels.BROADCAST)
    void broadcastGreet(String msg);

    @Gateway(requestChannel = ProducerChannels.DIRECT)
    void directGreet(String msg);
}

@RestController
class GreetingProducer {

    private final GreetingGateway gateway;

    ④
    @Autowired
    GreetingProducer(GreetingGateway gateway) {
        this.gateway = gateway;
    }

    @RequestMapping(method = RequestMethod.GET, value = "/hi/{name}")
    ResponseEntity<?> hi(@PathVariable String name) {
        String message = "Hello, " + name + "!";
        this.gateway.directGreet("Direct: " + message);
        this.gateway.broadcastGreet("Broadcast: " + message);
        return ResponseEntity.ok(message);
    }
}

```

- ❶ @EnableBinding 注解启用 Spring Cloud Stream，就像之前那样。
- ❷ Spring 框架注册自定义组件的构造型注释，但 Spring Integration 也可以将接口定义转换为 bean，所以我们需要一个自定义注释来激活 Spring Integration 的组件扫描，以找到我们的声明性的基于接口的消息传递网关。
- ❸ @MessagingGateway 是 Spring Integration 支持的许多消息传递端点之一（作为迄今为止使用的 Java DSL 的替代方案）。网关中的每个方法都用 @Gateway 注释，在参数中指定哪个消息通道应该继续。在这种情况下，就好像我们将消息发送到一个通道上，并将参数 .send(Message <String>) 作为有效载荷来调用。
- ❹ 对于其他业务逻辑，GreetingGateway 只是一个常规 bean。

流消费者

另一方面，我们希望在接收消息后将其注销。我们将使用接口来创建通道。重申一下，通道的名称不必与生产者的名称一致——只需要有代理的目的地名称即可（如示例 10-11 所示）。

示例10-11 接收问候的通道

```
package stream.consumer;
```

```
import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;
```

```
public interface ConsumerChannels {
```

```
    String DIRECTED = "directed";
```

```
    String BROADCASTS = "broadcasts";
```

❶

```
    @Input(DIRECTED)
```

```
    SubscribableChannel directed();
```

```
    @Input(BROADCASTS)
```

```
    SubscribableChannel broadcasts();
```

```
}
```

- ❶ 这里唯一值得注意的是，这些通道是用 `@Input`（理所当然）注释的，并返回一个 `MessageChannel` 子类型，它支持订阅传入的消息 `SubscribableChannel`。

请记住，Spring Cloud Stream 中的所有绑定默认情况下都是发布—订阅模式。这与消费者组有直接的联系，可以达到排他性的效果。例如，有一个名为 `foo` 的组中有 10 个消费者实例，只有一个实例会看到有一条消息传递给它。在分布式系统中，我们不能保证服务总是在运行，所以我们将利用持久订阅来确保一旦消费者重新连接到代理消息即被重新传递（如示例 10-12 所示）。

示例10-12 消费者的 `application.properties`

```
spring.cloud.stream.bindings.broadcasts.destination=greetings-pub-sub ❶
```

```
spring.cloud.stream.bindings.directed.destination=greetings-p2p ❷
```

```
spring.cloud.stream.bindings.directed.group=greetings-p2p-group
```

```
spring.cloud.stream.bindings.directed.durableSubscription=true
```

```
server.port=0
```

```
spring.rabbitmq.addresses=localhost
```

- ❶ 这段代码大家应该很熟悉，因为我们刚刚在生产者中提到过。
- ❷ 在这里，我们像以前一样配置一个目的地，但是我们也给定向消费者指定一个专属的消费者组 `greetings-p2p-group`，其中所有活动消费者中只有一个节点将看到有一条消息传入。确保一旦消费者重新连接，代理将通过指定具有 `durableSubscription` 的绑定来存储和重新传送失败的消息。

最后，在示例 10-13 中，我们来看下基于 DSL 的 Spring Integration Java 消费者。你对它应该很熟悉。

示例10-13 由Spring Integration Java DSL驱动的消费者

```
package stream.consumer.integration;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.context.annotation.Bean;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.messaging.SubscribableChannel;
import stream.consumer.ConsumerChannels;

@SpringBootApplication
@EnableBinding(ConsumerChannels.class)
❶
public class StreamConsumer {

    public static void main(String args[]) {
        SpringApplication.run(StreamConsumer.class, args);
    }

    ❷
    private IntegrationFlow incomingMessageFlow(SubscribableChannel incoming,
        String prefix) {

        Log log = LogFactory.getLog(getClass());

        return IntegrationFlows
            .from(incoming)
            .transform(String.class, String::toUpperCase)
            .handle(
```



```

String.class,
(greeting, headers) -> {
    log.info("greeting received in IntegrationFlow (" + prefix + "): "
        + greeting);
    return null;
}).get();
}

@Bean
IntegrationFlow direct(ConsumerChannels channels) {
    return incomingMessageFlow(channels.directed(), "directed");
}

@Bean
IntegrationFlow broadcast(ConsumerChannels channels) {
    return incomingMessageFlow(channels.broadcasts(), "broadcast");
}
}

```

- ❶ 与前面一样，`@EnableBinding` 启用消费者通道。
- ❷ 这个 `@Configuration` 类定义了两个基本相同的 `IntegrationFlow` 流程：接收传入的消息，通过大写转换，然后记录。一个流监听广播的问候，另一个流监听直接的点对点的问候。通过在链的最后一个组件中返回 `null` 来停止处理。

尝试一下。运行生产者节点的一个实例（无论哪个），并运行消费者的三个实例。访问 <http://localhost:8080/hi/World> 并观察消费者日志，看它们三个是否都将消息发送到广播通道；其中一个（没有说发送到哪个通道，所以检查所有的控制台）将消息发送到直接通道。杀死所有消费者节点，然后访问 <http://localhost:8080/hi/Again> 来提高破坏性。虽然所有的消费者都关闭了，但是因为我们指定了点对点连接是持久的，所以当你重新启动一个用户时就会看到有消息到达并记录到控制台。

总结

消息传递为复杂的分布式交互提供了一个消息通道。在后面的章节中我们将看到，可以在消息传递基础上连接批处理解决方案、工作流引擎和服务。

批处理和任务

云给我们带来了前所未有的规模效应。为了满足需求，启动新的应用程序实例几乎不需要任何成本，一旦应用程序扩张的势头褪去，又很容易缩减规模。这意味着，只要现在的工作能够实现并行化，我们就可以通过规模提高效率。还有一些是令人尴尬的并行问题：有些问题不需要节点之间协调，而有些可能需要一些协调。这两种类型的工作负载都是云计算环境的理想选择，还有一些工作负载本身就是串行的。对于没有特别并行化的工作，云计算环境对于将计算水平扩展到多个节点都是理想的。在本章中，我们将了解几种不同的新旧方式，以使用微服务来获取和处理数据。

批处理工作

批处理历史悠久。批处理是指一个程序同时处理批量输入的数据。历史上，批处理是利用计算资源的更有效的方式。这种方法通过确定交互式工作窗口的优先顺序来分摊一组机器的成本，如当操作员正在使用机器时，在晚上时间非交互式工作时，当机器空闲时。在当今的云时代，几乎无限的计算能力、高效的机器利用率并不是云计算适合批量处理的特别令人信服的理由。

使用大型数据集时批处理也很有吸引力。顺序数据——SQL 数据、.csv 文件等特别适用于批量处理。昂贵的资源，如文件、SQL 表游标和事务可能会被保留在一大块数据上，使得处理能够更快进行。

批处理支持名为窗口的逻辑概念——一个上限和一个下限，用来分隔一组数据。窗口可以是基于时间的：最近 60 分钟的所有记录，或者过去 24 小时的所有记录。也许这个窗口是合乎逻辑的：前 1000 个记录，或所有具有某种属性的记录。

如果要处理的数据集太大而不能放入内存，那么可以用更小的块来处理。块（chunk）是

一批数据的高效（尽管以资源为中心）划分。假设你想要访问规模在 2000 万行的产品销售数据库中的每条记录。没有分页，执行 `SELECT * FROM PRODUCT_SALES` 可能会导致整个数据集被加载到内存中，这可能会很快压垮系统。对这个大型数据集分页后效率要高得多，它一次只能载入一千（或一万）条记录。按顺序或并行处理块，向前移动，直到最终访问大型查询的所有记录，而无须同时将所有内容加载到内存中。

如果你的系统能够容忍陈旧的数据，那么批处理可以提高处理效率。这样的系统有很多，例如，直到本周末才会开始计算本周的报告，而不需要计算上周的销售情况。

Spring Batch

Spring Batch 是一个旨在支持处理大量记录的框架。Spring Batch 包括日志记录 / 跟踪、事务管理、作业处理统计、作业重启、跳过和资源管理。它已经成为 JVM 批处理的行业标准。

Spring Batch 的核心是 `job` 的概念，而 `job` 又可能有多个步骤。然后，每一步都会为可选的 `ItemReader`、`ItemProcessor` 和 `ItemWriter` 提供上下文（见图 11-1）。

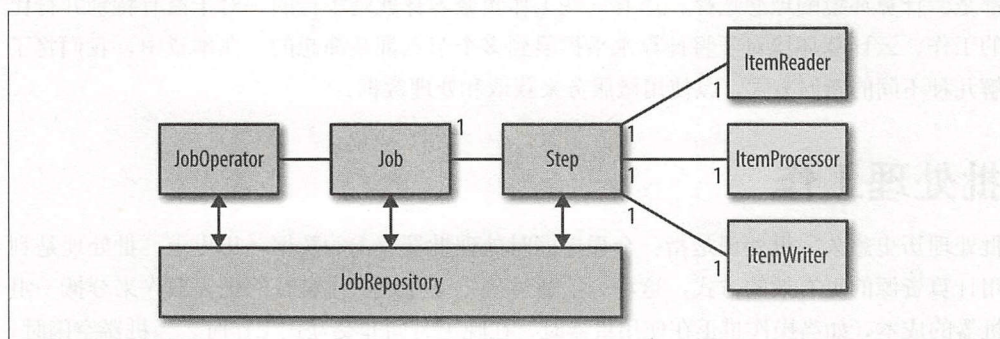


图11-1 Spring Batch作业的域

批处理作业有多个步骤。步骤指在数据发送到下一步之前进行某种准备或分段作业。你可以使用路由逻辑指导数据如何从一个步骤到下一个步骤：条件、并发和基本循环。步骤也可以在 `tasklet` 中定义通用业务功能。在该示例中，我们使用 Spring Batch 编排执行的序列（sequence）。步骤可以通过扩展 `ItemReader`、`ItemProcessor` 和 `ItemWriter` 以实现更多定义的处理。

`ItemReader` 接受外部的输入（.csv 或 XML 文档、SQL 数据库、目录等），并将其适配为可以在逻辑上工作的东西：项目（*item*）。项目可以是任何东西。项目可以是数据库中的记录，也可以是来自文本文件的段落、.csv 文件的记录或 XML 文档的节。Spring

Batch 提供了即开即用的 `ItemReader` 实现。`ItemReader` 实现一次读取一个项目，但是将结果项目存储在与指定的 `chunk` 大小匹配的缓冲区中。

如果指定了 `ItemProcessor`，那么将从 `ItemReader` 中给出每个项目。`ItemProcessor` 是做处理和转换的：数据输入、数据输出。`ItemProcessor` 是暂存或验证通用业务逻辑理想的地方，而不是专门做输入和输出。

如果指定了 `ItemWriter`，那么它将被指定给来自 `ItemProcessor`（如果这样配置）或 `ItemReader` 的累积（不是单个）项目。`ItemWriter` 将累积的项目块写入资源，如数据库或文件。`ItemWriter` 每次写入时都会尽可能多地写入数据，直到达到指定的块大小。

批处理大概就是这样。对于大多数类型的 I/O 来说在写入批处理项目时更高效。在缓冲区中，在写入批处理行时更高效，在批量写入数据库时效率也会更高。

我们的第一个批处理作业

首先来看一个 Job 和配置 Step 实例的流程（如示例 11-1 所示）。

示例11-1 具有三个步骤实例的简单作业

```
package processing;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.HttpStatusException;
import processing.email.InvalidEmailException;

import java.util.Map;

@Configuration
class BatchConfiguration {

    ❶ @Bean
    Job etl(JobBuilderFactory jbf, StepBuilderFactory sbf,
        Step1Configuration step1, ❷
        Step2Configuration step2, Step3Configuration step3) throws Exception {

        Step setup = sbf.get("clean-contact-table").tasklet(step1.tasklet(null)) ❸
            .build();
```



```

Step s2 = sbf.get("file-db").<Person, Person>chunk(1000)
    ④
    .faultTolerant()
    ⑤
    .skip(InvalidEmailException.class).retry(HttpStatusNotFoundException.class)
    .retryLimit(2).reader(step2.fileReader(null)) ⑥
    .processor(step2.emailValidatingProcessor(null)) ⑦
    .writer(step2.jdbcWriter(null)) ⑧
    .build();

    ⑨
Step s3 = sbf.get("db-file")
    .<Map<Integer, Integer>, Map<Integer, Integer>>chunk(100)
    .reader(step3.jdbcReader(null)).writer(step3.fileWriter(null)).build();

return jbf.get("etl").incrementer(new RunIdIncrementer()) ⑩
    .start(setup) ⑪
    .next(s2).next(s3).build(); ⑫
}
}

```

- ① 使用 `JobBuilderFactory` 和 `StepBuilderFactory` 定义一个 Spring Batch Job。
- ② 每个步骤还需要 bean，所以在配置类中定义了它们，并且注入到易于解引用的地方。
- ③ 第一个 Step 将使用 `Tasklet`——一种自由形式的回调，可以任何事情。通过将其包含在 Step 中来描述其相对于其他作业的执行情况。
- ④ 我们希望一次写入 10 条记录到配置的 `ItemWriter` 中。
- ⑤ 我们希望处理可能遇到的失败，所以配置了作业跳过策略（应该跳过单个记录的例外情况）和重试策略（针对给定项目应重复哪些步骤以及重复次数）。Spring Batch 在这里给你提供了很多的控制手段，这样就不必因为单个错误数据记录而中止整个工作。
- ⑥ 在 `Step1Configuration` 中取消引用 `FlatFileItemReader`。ItemReader 从 .csv 文件中读取数据并将其转换为 Person POJO。@Bean 定义需要参数，但 Spring 会提供这些参数。可以使用 null 参数调用方法，因为 bean 已经被创建（使用容器提供的参数），并且方法的返回值将会被缓存（作为预先创建的实例）。
- ⑦ `ItemProcessor` 是插入 Web 服务调用来验证 Person 记录中的电子邮件是否有效的理想位置。

- ⑧ 解引用 `ItemWriter` 来写入 `JDBCDataSource`。
- ⑨ 这一步从第一步开始查询刚刚保留的数据，计算记录具有给定年龄的频率，然后将这些计算写入输出 `.csv`。
- ⑩ `Spring Batch Job` 是参数化的。这些参数确认了 `job` 的标识。此标识保存在数据库的元数据表中。在这个例子中，元数据表保存在 `MySQL` 中。如果我们使用相同的参数启动同一个 `Job` 两次，那么该 `Job` 将拒绝运行，因为它已经观察到之前的作业，并在元数据表中做了记录。在这里，我们配置了一个 `RunIdIncrementer`，其用来增加一个现有的参数（如 `run.id`）的值，如果该参数存在，则派生一个新的 `key`。
- ⑪ 现在我们把作业中的步骤流程串起来，从 `setup` 步骤开始，继续执行步骤 `s1` 和步骤 `s3`。
- ⑫ 最后，建立 `Job`。

假如，我们运行这个作业时出错了：读取失败，工作中止，`Spring Batch` 将运行配置中的重试和跳过策略，并尝试继续运行。如果失败，其作业进度将被记录在元数据表中，然后操作员可以决定介入或重新启动（而不是启动另一个实例）作业。该作业需要从停止的地方恢复。这是可能的，因为有些 `ItemReader` 实现可以读取有进度记录的有状态资源，比如文件。

该 `Job` 不是很特别。按照配置，它将从单线程的 `ItemReader` 中串行读取。我们可以为 `Job` 配置一个 `TaskExecutor` 实现，并且 `Spring Batch` 将同时读取，通过配置的 `TaskExecutor` 支持的多个线程有效地划分读取时间。



你可以用记录的偏移量来重试失败的作业或使用 `TaskExecutor` 来并行读取，但不能同时做这两个事情！这是因为偏移量存储在本地线程中，最终在其他线程中观察到的值会被破坏。

`Spring Batch` 是有状态的，它为数据库中运行的所有作业保留元数据表。数据库记录，`job` 运行了多久，退出代码是什么，是否跳过某些行（以及是否中止了这个 `job`，还是只是跳过了）。运维人员（或自治机构）可以使用这些信息来决定重新运行或手动干预该作业。

让所有的东西都运行起来很简单。当 `Spring Batch` 的 `Spring Boot` 自动配置功能启动时，它会查找一个 `DataSource`，并尝试根据类路径上的模式自动创建适当的元数据表（如示例 11-2 所示）。

示例11-2 MySQL中的Spring Batch元数据表

```
mysql> show tables;
```

```
+-----+
| Tables_in_batch |
+-----+
| BATCH_JOB_EXECUTION |
| BATCH_JOB_EXECUTION_CONTEXT |
| BATCH_JOB_EXECUTION_PARAMS |
| BATCH_JOB_EXECUTION_SEQ |
| BATCH_JOB_INSTANCE |
| BATCH_JOB_SEQ |
| BATCH_STEP_EXECUTION |
| BATCH_STEP_EXECUTION_CONTEXT |
| BATCH_STEP_EXECUTION_SEQ |
+-----+
9 rows in set (0.00 sec)
```

在这个例子中，我们配置了两个容错级别：在被认为是错误之前，某个步骤可以重试两次。我们使用第三方的网络服务，它可能会不可用。你可以通过关闭机器的网络连接来模拟服务的可用性。你会发现其将无法连接，抛出一个 `HttpStatusCodeException` 子类，然后重试。我们还想跳过未经验证的记录，所以配置了一个跳过策略，只要抛出一个可以分配给 `InvalidEmailException` 类型的异常，就跳过该行的处理。

我们来检查个别步骤的配置。首先是 `setup` 步骤（如示例 11-3 所示）。

示例11-3 配置第一个步骤，其中只有一个Tasklet

```
package processing;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
```

```
@Configuration
```

```
class Step1Configuration {
```

```
    @Bean
```

```
    Tasklet tasklet(JdbcTemplate jdbcTemplate) {
```

```
        Log log = LogFactory.getLog(getClass());
```

```

return (contribution, chunkContext) -> { ❶
    log.info("starting the ETL job.");
    jdbcTemplate.update("delete from PEOPLE");
    return RepeatStatus.FINISHED;
};
}
}

```

- ❶ Tasklet 是一个通用的回调函数，你可以在其中进行任何操作。在这种情况下，我们通过删除任何数据来对 PEOPLE 表进行排序。

Tasklet 为第二个步骤规划表格，这一阶段用来从输入文件读取数据，验证电子邮件，然后将结果写入新清理的数据库表。这一步从一个 .csv 文件中提取数据，这个文件有三列：人物的姓名、年龄和电子邮件（如示例 11-4 所示）。

示例 11-4 要提取的.csv文件的内容

```

tam mie,30,tammie@email.com
srinivas,35,srinivas@email.com
lois,53,lois@email.com
bob,26,bob@email.com
jane,18,jane@email.com
jennifer,20,jennifer@email.com
luan,34,luan@email.com
toby,24,toby@email.com
toby,24,toby@email.com
...

```

这一步演示如何配置一个典型的 ItemReader、ItemProcessor 和 ItemWriter（如示例 11-5 所示）。

示例 11-5 读取.csv文件中的记录并将其加载到数据库表中

```

package processing;

import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;
import processing.email.EmailValidationService;

```



```

import processing.email.InvalidEmailException;

import javax.sql.DataSource;

@Configuration
class Step2Configuration {

    @Bean
    @StepScope
    ❶ FlatFileItemReader<Person> fileReader(
        @Value("file://#{jobParameters['input']}") Resource in) ❷
        throws Exception {

        ❸ return new FlatFileItemReaderBuilder<Person>().name("file-reader")
            .resource(in).targetType(Person.class).delimited().delimiter(",")
            .names(new String[] { "firstName", "age", "email" }).build();
    }

    @Bean
    ItemProcessor<Person, Person> emailValidatingProcessor(
        EmailValidationService emailValidator) { ❹
        return item -> {
            String email = item.getEmail();
            if (!emailValidator.isEmailValid(email)) {
                throw new InvalidEmailException(email);
            }
            return item;
        };
    }

    @Bean
    JdbcBatchItemWriter<Person> jdbcWriter(DataSource ds) { ❺
        return new JdbcBatchItemWriterBuilder<Person>()
            .dataSource(ds)
            .sql(
                "insert into PEOPLE( AGE, FIRST_NAME, EMAIL)"
                + " values (:age, :firstName, :email)".beanMapped().build();
    }
}

```

- ❶ 使用 `@StepScope` 注释的 bean 不是单例。每次运行作业实例时都会重新创建它们。
- ❷ `@Value` 使用 Spring 表达式语言从 Spring Batch `jobParameters` 上下文中获取 `input` 作业参数。这是批处理作业中的一种常见做法：你可以今天运行指向反映当前日期

的文件的作业，明天再使用不同日期的文件作业。

- ❸ Spring Batch 的 Java 配置 DSL 提供了方便的构建器 API 来配置常见的 `ItemReader` 和 `ItemWriter` 实现。在这里，我们配置了一个 `ItemReader`，它从 `in` 文件中读取一个以逗号分隔的行，并将这些列映射到姓名，然后映射到 `PersonPOJO` 上的字段。然后将这些字段重叠在 `Person` 的 `POJO` 实例上，并且从 `ItemReader` 中返回 `Person` 来进行累加。
- ❹ 自定义的 `ItemProcessor` 只是委托给 `EmailValidationService` 实现，而后者调用 REST API。
- ❺ `JdbcBatchItemWriter` 接受 `ItemProcessor<Person, Person>` 的结果，并使用我们提供的 SQL 语句将其写入基础 SQL 数据存储中。SQL 语句的命名参数对应于从 `ItemProcessor` 生成的 `Person POJO` 实例上的 `JavaBean` 属性。

提取完成。在最后一步，分析结果，确定提取记录中给定年龄的分布（如示例 11-6 所示）。

示例11-6 分析数据并将结果写入输出文件

```
package processing;
```

```
import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.item.database.JdbcCursorItemReader;
import org.springframework.batch.item.database.builder.JdbcCursorItemReaderBuilder;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemWriterBuilder;
import org.springframework.batch.item.file.transform.DelimitedLineAggregator;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;
```

```
import javax.sql.DataSource;
import java.util.Collections;
import java.util.Map;
```

```
@Configuration
```

```
class Step3Configuration {
```

```
❶
```

```
@Bean
```

```
JdbcCursorItemReader<Map<Integer, Integer>> jdbcReader(DataSource dataSource) {
    return new JdbcCursorItemReaderBuilder<Map<Integer, Integer>>()
        .dataSource(dataSource)
        .name("jdbc-reader")
```



```

        .sql("select COUNT(age) c, age a from PEOPLE group by age")
        .rowMapper(
            (rs, i) -> Collections.singletonMap(rs.getInt("a"), rs.getInt("c")))
        .build();
    }

    ②
    @Bean
    @StepScope
    FlatFileItemWriter<Map<Integer, Integer>> fileWriter(
        @Value("file://#{jobParameters['output']}") Resource out) {
        //@formatter:off
        DelimitedLineAggregator<Map<Integer, Integer>> aggregator =
            new DelimitedLineAggregator<Map<Integer, Integer>>() {
            {
                setDelimiter(",");
                setFieldExtractor(ageAndCount -> {
                    Map.Entry<Integer, Integer> next = ageAndCount.entrySet().iterator()
                        .next();
                    return new Object[] { next.getKey(), next.getValue() };
                });
            }
        };
        //@formatter:on

        return new FlatFileItemWriterBuilder<Map<Integer, Integer>>()
            .name("file-writer").resource(out).lineAggregator(aggregator).build();
    }
}

```

- ❶ JdbcCursorItemReader 执行查询，然后访问结果集中的每个结果。使用与 Spring 框架的 JdbcTemplate 相同的 RowMapper<T>，它将每一行映射到处理器或写入器所需的对象——一个反映年龄的键 Map<Integer,Integer>，并计算结果集中年龄的频率。
- ❷ 写入器是有状态的，并且需要在每次运行 Job 的时候重新创建，因为每个作业都将结果写入不同名字的文件。FlatFileItemWriter 需要一个 LineAggregator 实例来弄清楚如何将传入的 POJO (Map <Integer,Integer>) 转换成行并写入 .csv 输出文件。

现在只需要运行它！Spring Boot 的自动配置功能在启用时默认会运行这个 job。这个 job 需要参数 (input 和 output)，所以我们禁用了默认的行为，并在 CommandLineRunner 实例中显式启动了这个 job (如示例 11-7 所示)。

示例11-7 批处理应用程序的入口点

```
package processing;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.web.client.RestTemplate;

import javax.sql.DataSource;
import java.io.File;

@EnableBatchProcessing
@SpringBootApplication
public class BatchApplication {

    public static void main(String[] args) {
        SpringApplication.run(BatchApplication.class, args);
    }

    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @Bean
    JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    ❶
    @Bean
    CommandLineRunner run(JobLauncher launcher, Job job,
        @Value("${user.home}") String home) {
        return args -> launcher.run(job,
            new JobParametersBuilder().addString("input", path(home, "in.csv"))
                .addString("output", path(home, "out.csv")).toJobParameters());
    }

    private String path(String home, String fileName) {
```



```
    return new File(home, fileName).getAbsolutePath();  
}  
}
```

- ❶ `CommandLineRunner` 在应用程序启动时运行。在其中，我们只是硬编码指向本地文件系统的路径引用，并将其作为 `JobParameter` 实例传递。

`Job` 默认同步运行。`JobLauncher#run` 方法返回一个 `JobExecution`，一旦其运行完成就可以用来询问工作状态。如果配置的 `JobLauncher` 具有 `TaskExecutor`，则可以异步运行该作业。

Spring Batch 旨在安全地处理大量的数据，尽管到目前为止我们只看到了一个节点。稍后，我们将看到如何使用通过消息传递的远程 Spring 批处理作业分区来处理 Spring 批处理作业。

Spring Cloud Task 是一个通用抽象，用于管理（并使其可观察）进程从运行到终止的整个生命周期。稍后我们会看到，它可以与任何基于 Spring Boot 的服务一起工作，该服务定义了 `CommandLineRunner` 或 `ApplicationRunner` 的实现，它们都是简单的回调接口。当它们被 Spring bean 从 `main(String args [])` 方法的应用程序的 `String [] args` 数组回调时，Spring Boot 会自动配置一个 `CommandLineRunner`，它将在 Spring 应用程序上下文中运行任何现有 Spring Batch Job 的实例。所以我们的 Spring Batch 作业已经成为主要的候选者，可以像 Spring Cloud Task 那样被运行和管理。在讨论任务管理时，我们会更多地考虑这个问题。

调度

我们常常会想：“我该如何安排这些 job？”如果你的 job 是基于 `fat-jar` 可部署的，则可以从环境配置源（比如命令行参数或者环境变量）中获取配置；如果你只有一个节点，那么只要用好 `cron` 就足够了。

如果你希望对 job 调度的方式进行更细粒度的控制，那么可以在 JDK 中使用 `ScheduledExecutorService`，甚至可以稍微提升一下抽象，利用 Spring 的 `@Scheduled` 注解，其然后委派给 `java.util.concurrent.Executor` 实例。这种方法的主要缺陷是没有记录 job 是否已经运行，没有内置的集群概念。如果 worker 节点死掉了会怎么样？作业会在另一个 worker 节点上重新启动吗？如果死掉的节点应该以某种方式恢复呢？如何避免经典的脑裂问题——两个节点都假设自己是 leader，最终同时运行？

有些商业的调度器，如 BMC、Flux Scheduler 和 Autosys。这些工具十分强大，可以在集群中调度任何类型的工作负载。这些工具如何在云环境中运行可能没有你想象的

那么简单。如果你想更好地控制 job 的调度和生命周期，你可以查看 Spring 与 Quartz Enterprise Job Scheduler (<http://www.quartz-scheduler.org/>) 的集成。Quartz 可以很好地在集群环境中运行，应该可以满足作业调度的需求。而且它也是开源的，很容易在云环境中运行。

另一种方法可能是使用 *leader* 选举来管理集群中 leader 节点的升级和降级。Leader 节点需要是有状态的，否则会有存在运行相同的工作的风险。Spring Integration 有一个支持 leader 选举和分布式锁的抽象，将其委托给 Apache Zookeeper、Hazelcast 或其他组件来实现。它使事务性地将一个节点从 leader 转移到另一个节点变得轻而易举。它提供了与 Apache Zookeeper、Hazelcast 和 Redis 协同工作的实现。这样 leader 节点将按照时间表将作业集中到集群中的其他节点。节点间的通信可以像消息传递一样简单。我们将在本章的稍后部分探讨消息传递。

答案不止一个，但值得强调的是，这个问题并没有什么新意，因为当今的大多数工作负载是事件驱动的，而不是时间驱动的。

通过消息传递远程分区 Spring 批处理作业

我们在前面已经看到了，TaskExecutor 可以使在 Spring 批处理作业中并行读取变得更容易。Spring Batch 还支持通过两种机制来并行化写入：远程分区 (*remote partitioning*) 和远程分块 (*remote chunking*)。

在功能上，远程分区和远程分块将单个步骤的控制转发到集群中的另一个节点，通常通过消息传递基质（通过 Spring Framework MessageChannel 实例连接）进行连接。

远程分区将消息发布到另一个节点上，这个节点包含读取记录的范围（0~100，100~200 等），运行实际的 ItemReader、ItemProcessor 或 ItemWriter。这种方法要求 worker 节点可以访问 leader 节点的所有资源。例如，worker 节点要读取文件中的一系列记录，则需要访问该文件。从配置的 worker 节点返回的状态由 leader 节点聚合。所以，如果你的 job 在 ItemReader 和 ItemWriter 中是 I/O 绑定的，那么首选远程分区。

远程分块类似于远程分区，不同的一点是在 leader 节点上读取数据并通过线路将数据发送到 worker 节点进行处理。然后将处理好的结果发送回 leader 节点写入。所以如果你的 job 在 ItemProcessor 中是 CPU 绑定的，那么选择远程分块会更好。

远程分块和远程分区都增强了像 Cloud Foundry 这样的平台的弹性。你可以增加尽可能多的节点来处理作业，作业完成后再停掉那些节点。



Spring Batch 主管 Michael Minella 在 2012 年芝加哥 Java 用户组 (<https://www.youtube.com/watch?v=CYTj5YT7CZU>) 讨论会中详细解释了这一切。

批处理几乎总是 I/O 绑定的，所以我们发现，比起远程分块来说，更多的应用程序使用的是远程分区（虽然你的情况可能会有所不同）。我们来看一个例子。为了使远程分区能够工作，leader 节点和 worker 节点都需要访问 Spring Batch JobRepository，它拥有 Spring Batch 不同实例背后的状态。JobRepository 反过来需要一些其他的组件。因此，任何远程分区应用程序都将具有仅存在于 worker 节点上的代码、仅存在于 leader 节点上的代码以及存在于两者之中的代码。值得庆幸的是，Spring 提供了一个自然的机制来在一个单独的代码库中执行这个划分：profile（配置文件）！我们可以在配置文件中为某些对象添加标签，并在运行时有条件地打开或关闭这些对象（如示例 11-8 所示）。

示例 11-8 我们为这个应用程序定义了一组配置文件

```
package partition;
```

```
public class Profiles {
```

```
    public static final String WORKER_PROFILE = "worker"; ❶
```

```
    public static final String LEADER_PROFILE = "!" + WORKER_PROFILE; ❷  
}
```

❶ worker 节点的配置文件。

❷ 只要 worker 配置文件不是活动的，leader 的配置文件就是活动的。

在远程分区应用程序中，有一个或多个 worker 节点和一个 leader 节点。leader 节点使用消息传递协调 worker 节点。具体而言，它通过 MessageChannel 实例与 worker 节点通信，可能是连接到 RabbitMQ 或 Apache Kafka 等消息代理的 Spring Cloud Stream 支持的 MessageChannel 实例。

我们来看一个简单的 Spring Batch Job，该作业读取一个表（我们的 PEOPLE 表）中的所有记录，并将它们复制到另一个名为 NEW_PEOPLE 的空表中。这只是一个简单的例子，所以我们不想被其他的东西分散注意力，而是专注于实现解决方案所需的组件。

应用程序的入口点激活了 Spring Batch 和 Spring Integration，并配置了一个 JdbcTemplate 和一个默认的轮询器，用于定期轮询下游组件（如示例 11-9 所示）。

示例11-9 应用程序的入口点

```
package partition;

import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.integration.annotation.IntegrationComponentScan;
import org.springframework.integration.dsl.core.Pollers;
import org.springframework.integration.scheduling.PollerMetadata;
import org.springframework.jdbc.core.JdbcTemplate;

import javax.sql.DataSource;
import java.util.concurrent.TimeUnit;
```

```
@EnableBatchProcessing
```

❶

```
@IntegrationComponentScan
```

```
@SpringBootApplication
```

```
public class PartitionApplication {

    public static void main(String args[]) {
        SpringApplication.run(PartitionApplication.class, args);
    }
}
```

❷

```
@Bean(name = PollerMetadata.DEFAULT_POLLER)
```

```
PollerMetadata defaultPoller() {
    return Pollers.fixedRate(10, TimeUnit.SECONDS).get();
}
```

```
@Bean
```

```
JdbcTemplate jdbcTemplate(DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
```

```
}
```

❶ 激活 Spring Batch 和 Spring Integration。

❷ 为需要轮询的 MessageChannel 实现指定一个默认的全局轮询器。

该应用程序有一个单一的 Spring Batch Job，其中包含两个 Step。第一个 Step 是 Tasklet 通过清空数据库（MySQL 中的 truncate）来分阶段执行数据库（如示例 11-10 所示）。第二个 Step 是分区，其配置我们将在示例 11-11 中看到。

示例11-10 分区Step配置

```
package partition;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile(Profiles.LEADER_PROFILE)
1 class JobConfiguration {

    @Bean
    Job job(JobBuilderFactory jbf, LeaderStepConfiguration lsc) {
        return jbf.get("job").incrementer(new RunIdIncrementer())
            .start(lsc.stagingStep(null, null)) 2
            .next(lsc.partitionStep(null, null, null, null)) 3
            .build();
    }
}
```

❶ Job 只存在于 leader 节点配置文件中。

❷ Tasklet 重置 NEW_PEOPLE 表。

❸ 划分的 Step 将作业指派到其他节点。

分区的 Step 在 leader 节点上不做任何工作。相反，它充当一种代理节点，将工作分派给 worker 节点。分区的 Step 需要知道有多少 worker 节点(网格大小)可用。在示例 11-10 中，我们已经通过网格大小的属性设置了一个硬编码的值，但是可以通过使用 Spring Cloud 的 `DiscoveryClient` 抽象，动态地查找服务实例来动态解析 worker 节点池的大小，或通过询问底层平台的 API（例如 Cloud Foundry API Client，<https://docs.cloudfoundry.org/buildpacks/java/java-client.html>）来获知其大小（如示例 11-11 所示）。

示例11-11 分区步骤的配置

```
package partition;

import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.explore.JobExplorer;
import org.springframework.batch.core.partition.PartitionHandler;
```

```

import org.springframework.batch.core.partition.support.Partitioner;
import org.springframework.batch.integration.partition.MessageChannelPartitionHandler;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.core.MessagingTemplate;
import org.springframework.jdbc.core.JdbcOperations;
import org.springframework.jdbc.core.JdbcTemplate;

```

`@Configuration`

```

class LeaderStepConfiguration {

```

①

`@Bean`

```

Step stagingStep(StepBuilderFactory sbf, JdbcTemplate jdbc) {

```

```

    return sbf.get("staging").tasklet((contribution, chunkContext) -> {

```

```

        jdbc.execute("truncate NEW_PEOPLE");

```

```

        return RepeatStatus.FINISHED;

```

```

    }).build();

```

```

}

```

②

`@Bean`

```

Step partitionStep(StepBuilderFactory sbf, Partitioner p, PartitionHandler ph,

```

```

    WorkerStepConfiguration wsc) {

```

```

    Step workerStep = wsc.workerStep(null);

```

```

    return sbf.get("partitionStep").partitioner(workerStep.getName(), p)

```

```

        .partitionHandler(ph).build();

```

```

}

```

③

`@Bean`

```

MessageChannelPartitionHandler partitionHandler(

```

```

    @Value("${partition.grid-size:4}") int gridSize,

```

```

    MessagingTemplate messagingTemplate, JobExplorer jobExplorer) {

```

```

    // @formatter:off

```

```

    MessageChannelPartitionHandler ph =

```

```

        new MessageChannelPartitionHandler();

```

```

    // @formatter:on

```

```

    ph.setMessagingOperations(messagingTemplate);

```

```

    ph.setJobExplorer(jobExplorer);

```

```

    ph.setStepName("workerStep");

```

```

    ph.setGridSize(gridSize);

```

```

    return ph;

```

```

}

```



```

4
@Bean
MessagingTemplate messagingTemplate(LeaderChannels channels) {
    return new MessagingTemplate(channels.leaderRequestsChannel());
}

5
@Bean
Partitioner partitioner(JdbcOperations jdbcTemplate,
    @Value("${partition.table:PEOPLE}") String table,
    @Value("${partition.column:ID}") String column) {
    return new IdRangePartitioner(jdbcTemplate, table, column);
}
}

```

- ❶ Leader 节点上的 Job 中的第一个 Step 重置数据库。
- ❷ 下一步，分区步骤需要知道在远程节点上调用哪个 worker 步骤，Partitioner 还是 PartitionHandler。
- ❸ PartitionHandler 负责在 leader 节点上获取原始的 StepExecution，并将其分割成一个 StepExecution 实例的集合，以便将其发布到 worker 节点。PartitionHandler 想要知道有多少个 worker 节点，以便它可以相应地分配工作。这个特殊的 PartitionHandler 实现使用 MessageChannel 实例与工作节点协调。
- ❹ 因此需要一个 MessagingOperations 实现方便地发送或接收消息。
- ❺ Partitioner 负责将工作负载分区。在本示例中，分区是通过原始表 PEOPLE 中记录的 ID 的一些基本划分来完成的。在这个例子中的所有代码中，这可能是最乏味的，它通常与你的具体业务领域和应用程序相关。

worker 节点从代理中获取工作，并将请求路由到一个 StepExecutionRequestHandler 上，然后使用 StepLocator 来解析 worker 应用程序上下文中的实际步骤(如示例 11-12 所示)。

示例11-12 分区步骤的配置

```

package partition;

import org.springframework.batch.core.explore.JobExplorer;
import org.springframework.batch.core.step.StepLocator;
import org.springframework.batch.integration.partition.BeanFactoryStepLocator;
import org.springframework.batch.integration.partition.StepExecutionRequest;
import org.springframework.batch.integration.partition.StepExecutionRequestHandler;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

```

```
import org.springframework.context.annotation.Profile;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.support.GenericHandler;
import org.springframework.messaging.MessageChannel;
```

```
@Configuration
```

```
@Profile(Profiles.WORKER_PROFILE)
```

❶

```
class WorkerConfiguration {
```

❷

```
@Bean
```

```
StepLocator stepLocator() {
```

```
    return new BeanFactoryStepLocator();
```

```
}
```

❸

```
@Bean
```

```
StepExecutionRequestHandler stepExecutionRequestHandler(JobExplorer explorer,
```

```
    StepLocator stepLocator) {
```

```
    StepExecutionRequestHandler handler = new StepExecutionRequestHandler();
```

```
    handler.setStepLocator(stepLocator);
```

```
    handler.setJobExplorer(explorer);
```

```
    return handler;
```

```
}
```

❹

```
@Bean
```

```
IntegrationFlow stepExecutionRequestHandlerFlow(WorkerChannels channels,
```

```
    StepExecutionRequestHandler handler) {
```

```
    MessageChannel channel = channels.workerRequestsChannels();
```

```
    //@formatter:off
```

```
    GenericHandler<StepExecutionRequest> h =
```

```
        (payload, headers) -> handler.handle(payload);
```

```
    //@formatter:on
```

```
    return IntegrationFlows.from(channel)
```

```
        .handle(StepExecutionRequest.class, h)
```

```
        .channel(channels.workerRepliesChannels().get());
```

```
}
```

```
}
```

- ❶ 这些对象应该只存在于 worker 节点和 worker 配置文件中。
- ❷ StepLocator 负责按名称定位 Step 实现。在这个例子中，它将通过筛选包含的 BeanFactory 实现中的 bean 来解析 Step。

- ③ StepExecutionRequestHandler 是 worker 节点到 leader 节点上的 PartitionHandler 的对应部分：它接受传入的请求并将它们转换成给定的 Step 的执行，然后将结果发送回来。
- ④ StepExecutionRequestHandler 并不知道传入的 StepExecutionRequests 将到达的 MessageChannel 实例，所以我们使用 Spring Integration 触发 StepExecutionRequestHandler bean 的 #handle(StepExecutionRequest) 方法来响应消息，并确保返回值作为回复通道上的消息发送出去。

最后，在示例 11-13 中，激活实际步骤以及关于要读取哪些数据的信息。reader 可以访问与 leader 节点相同的状态（在本例中是一个 JDBC DataSource）是非常重要的。

示例 11-13 分区步骤的配置

```
package partition;
```

```
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.database.JdbcPagingItemReader;
import org.springframework.batch.item.database.Order;
import org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.database.support.MySqlPagingQueryProvider;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import javax.sql.DataSource;
import java.util.Collections;
```

```
@Configuration
```

```
class WorkerStepConfiguration {
```

①

```
@Value("${partition.chunk-size}")
private int chunk;
```

②

```
@Bean
```

```
@StepScope
```

```
JdbcPagingItemReader<Person> reader(DataSource dataSource,
    @Value("#{stepExecutionContext['minValue']}") Long min,
    @Value("#{stepExecutionContext['maxValue']}") Long max) {
```

```

MySQLPagingQueryProvider queryProvider = new MySQLPagingQueryProvider();
queryProvider
    .setSelectClause("id as id, email as email, age as age, first_name as firstName");
queryProvider.setFromClause("from PEOPLE");
queryProvider.setWhereClause("where id >= " + min + " and id <= " + max);
queryProvider.setSortKeys(Collections.singletonMap("id", Order.ASCENDING));

JdbcPagingItemReader<Person> reader = new JdbcPagingItemReader<>();
reader.setDataSource(dataSource);
reader.setFetchSize(this.chunk);
reader.setQueryProvider(queryProvider);
reader.setRowMapper((rs, i) -> new Person(rs.getInt("id"), rs.getInt("age"),
    rs.getString("firstName"), rs.getString("email")));
return reader;
}

```

```

③
@Bean
JdbcBatchItemWriter<Person> writer(DataSource ds) {
    return new JdbcBatchItemWriterBuilder<Person>()
        .beanMapped()
        .dataSource(ds)
        .sql(
            "INSERT INTO NEW_PEOPLE(age,first_name,email) VALUES(:age, :firstName, :email )"
        ).build();
}

④
@Bean
Step workerStep(StepBuilderFactory sbf) {
    return sbf.get("workerStep").<Person, Person>chunk(this.chunk)
        .reader(reader(null, null, null)).writer(writer(null)).build();
}
}

```

- ① 这是一个 Spring Batch 作业，所以 JdbcPagingItemReader 实现仍然关心记录分块。
- ② 请注意，我们使用的是 JdbcPagingItemReader 而不是像 JdbcCursorItemReader 这样的实现，因为我们没有能力跨所有 worker 节点共享 JDBC 结果游标。最终，两种实现都让我们将一个大的 JDBC ResultSet 分成更小的部分，而且知道何时何地应用哪个部分是很有用的。ItemReader 将在 PartitionHandler 的 ExecutionContext 中接收一个键和值的映射，提供有用的信息，在本示例中包括要读取的行的边界。由于这些值对 Step 的每次执行都是唯一的，因此 ItemReader 已经被赋予了 @ScopeScope。

③ JdbcBatchItemWriter 通过将 POJO 属性映射到 SQL 语句中的命名参数，将 Person POJO 的字段写入数据库。

④ 最后，我们建立 woker Step。这一点大家应该非常熟悉。

最后，应用程序组件通过 MessageChannel 实现连接。在这个示例中使用 Spring Cloud Stream 即 RabbitMQ 定义 leaderRequests、workerRequests 和 workerReplies。MessageChannel (leaderReplies- AggregatedChannel) 只用于连接内存中的两个组件。请你仔细阅读 LeaderChannels、WorkerChannels 和 application.properties 的源代码来查看这些定义。鉴于我们之前讨论过 Spring Cloud Stream，这里就不再赘述。

需要设置 PEOPLE 表，以及名为 NEW_PEOPLE 的相同配置的该表的重复版本。使用 worker 配置文件启动几个 woker 节点实例。然后通过不指定特定配置文件启动 leader 节点。这些都运行后，你可以启动一个 job 实例。这个例子不会在应用程序启动时运行 Job，但是你可以触发一个包含在源代码中的 REST 端点来运行：`curl -d{} http://localhost:8080/migrate`。

任务管理

Spring Boot 知道如何处理我们的 Job。当应用程序启动时，Spring Boot 运行所有 CommandLineRunner 实例，包括 Spring Boot 的 Spring Batch 自动配置提供的实例。然而，从外部看，没有合理的方法知道该如何运行这个 job。我们不一定知道它是不是描述了一个会终止并产生退出状态的工作负载。我们没有共同的基础设施来捕捉任务的开始和结束时间。如果发生异常，也没有基础设施来支持异常处理。Spring Batch 提供了这些概念，但是我们如何处理那些不是 Spring Batch Job 实例的东西，比如 CommandLineRunner 或 ApplicationRunner 实例呢？Spring Cloud Task 可以帮我们处理。Spring Cloud Task 提供了识别、执行和查询任务的方法！

任务 (task) 可以运行并具有预期的最终状态。任务是任何短暂进程或工作负载的理想抽象，它可能需要运行非常长的时间（比服务的主要请求和响应流程中的理想事务更长，这可能意味着几秒、几小时甚至几天）。任务描述一次（响应事件）或按计划运行的工作负载。常见的例子包括：

- 有人要求系统生成并发送重置密码电子邮件。
- 每当新文件到达目录时就运行一个 Spring 批处理作业。
- 定期进行垃圾文件收集和审核不一致的数据或消息队列日志的应用程序。
- 动态文档（或报告）生成。
- 媒体转码。

任务的一个关键特性是它提供了一个统一的参数化接口。Spring Cloud Task 任务接受参数和 Spring Boot 配置属性。如果你指定了参数，Spring Cloud Task 会将它们持久化为 CommandLine Runner 或 ApplicationRunner 参数或 Spring Batch JobParameter 实例。任务可以使用 Spring Boot 配置属性进行配置。事实上，我们稍后会看到，Spring Cloud Data Flow 甚至足够聪明，可以让你查询给定任务的已知属性列表，并为特定任务的属性渲染表单。



如果你希望自己的任务能够享受与 Spring Cloud Data Flow 巧妙集成的好处，则需要包含 Spring Boot 配置处理器（org.springframework.boot:spring-boot-configuration-processor）并定义一个 @ConfigurationProperties 组件。

我们来看一个简单的例子（见示例 11-14）。启动一个新的 Spring Boot 项目，然后添加 org.springframework.cloud:spring-cloud-task-starter。

示例11-14 一个独立的Spring Boot应用程序，它提供了Spring Cloud Task运行的CommandLineRunner package task;

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.task.configuration.EnableTask;
import org.springframework.cloud.task.repository.TaskExplorer;
import org.springframework.context.annotation.Bean;
import org.springframework.data.domain.PageRequest;

import java.util.stream.Stream;
```

❶

@EnableTask

@SpringBootApplication

public class HelloTask {

private Log log = LogFactory.getLog(getClass());

public static void main(String args[]) {

SpringApplication.run(HelloTask.class, args);

}

@Bean

CommandLineRunner runAndExplore(TaskExplorer taskExplorer) {


```

return args -> {
    Stream.of(args).forEach(log::info);

    ❷
    taskExplorer.findAll(new PageRequest(0, 1)).forEach(
        taskExecution -> log.info(taskExecution.toString()));
    };
}
}
}

```

- ❶ 启用 Spring Cloud Task。
- ❷ 注入 Spring Cloud Task TaskExplorer 来检查当前正在运行的任务的执行情况。

当运行这个例子的时候，CommandLineRunner 可以像我们预期的那样在基于非任务的应用程序中运行。不过，我们可以注入 TaskExplorer 来动态地查询正在运行的任务（或者任何任务）的状态。TaskExplorer 了解应用程序上下文中 CommandLineRunner、ApplicationRunner 和 Spring Batch 作业实例。通过 Spring Cloud Task 的批量集成（org.springframework.cloud:spring-cloud-task-batch）获得对 Spring Batch Job 工作负载的支持。

我们将在稍后讨论 Spring Cloud Data Flow 时再次讨论 Spring Cloud Task。

通过 Workflow 进行的以工作流为中心的整合

虽然 Spring Batch 为我们提供了基本的工作流功能，但其目的是支持处理大型数据集，而不是将自治代理和人工代理整合到一个全面的工作流中。工作流（Workflow）是通过自治代理系统（和人类）明确建模工作进度的实践。工作流（Workflow）系统定义了一个状态机，并构建了一个状态机向目标进程的建模。工作流系统被设计成既是技术工件又是更高级别业务流程的描述。



工作流与业务流程管理（business process management）和业务流程建模（business process modeling）的思想（两者都令人困惑地缩写为 BPM）重叠。BPM 指的是描述和运行工作流的技术能力，也指自动化业务的管理规则。分析师使用 BPM 来识别业务流程，他们使用 Activiti（<https://www.activiti.org>）等工作流引擎进行建模和执行。

工作流系统简化了建模流程。通常，工作流系统提供了便于可视化建模的设计工具。这样做的主要目的是为业务和技术人员提供一个最小程度的工件。一个流程（process）模型不是直接可执行的代码，而是一系列的步骤。开发者应该为各种状态提供适当的行为。

workflow 系统通常提供用来存储和查询各种流程的状态的方法。像 Spring Batch 一样， workflow 系统提供了一个内置机制来设计失败时的补偿行为。

从设计的角度来看， workflow 系统有助于保持你的服务和实体无状态，并且不会出现无关的流程状态。我们看到过很多系统，一个实体通过短暂的一次性进程在实体和数据库本身身上表现为布尔值 (is_enrolled.is_fulfilled 等)。这些标志将实体的设计搞得很混乱。

workflow 系统将角色映射到步骤序列。这些角色有点像 UML 中的泳道。像泳道这样的 workflow 引擎可以描述人工任务列表以及自治活动。

所有人都需要 workflow 吗？显然不是。我们已经看到，对于具有复杂流程的组织，或者受到法规和政策约束，需要查询流程状态的方法时 workflow 是最有用的。对于人力和服务朝着更大业务需求迈进的协作流程来说，这是最佳选择，例如贷款审批、法律合规、保险审查、文件修改和文件发布等。

workflow 简化了设计，将业务逻辑从所需状态释放出来，以支持流程的审计和报告。在本节中我们将看到，像批处理一样，它提供了一种解决复杂、多代理和多节点进程中的故障的有意义的方法。虽然 workflow 引擎本身不是一个 saga 执行协调员，但是可以在 workflow 引擎之上建立一个 saga 执行协调员。如果我们正确地使用它就会得到很多想不到的好处。稍后我们会看到， workflow 还可以通过消息传递基础设施在云环境上进行横向扩展。

我们来做一个简单的练习。假设你有一个新用户的注册流程。业务将新注册的进展作为一个指标。从概念上讲，注册流程是一件简单的事情：用户通过一个表单提交一个新的注册请求，然后必须验证，如果有错误，则修复。一旦表格被正确接受，必须发送确认电子邮件。用户必须单击确认电子邮件并触发一个已知的端点，确认电子邮件是可达的。用户可以在一分钟或两周或一年内做这件事！长期事务仍然有效。我们可以使流程更加复杂，并在 workflow 的定义中指定超时和上报。然而，现在，这是一个足够有趣的例子，涉及自动和人力的工作，以实现新用户注册的目标（如图 11-2 所示）。

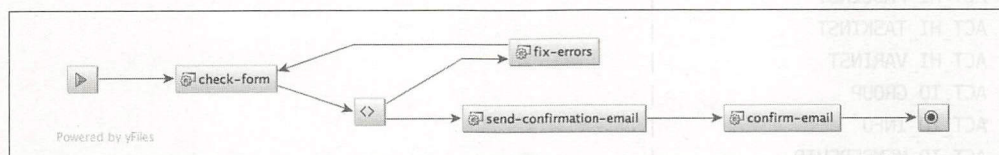


图11-2 在IntelliJ IDEA的yFiles驱动的BPMN 2.0预览中查看使用BPMN 2.0建模的注册流程

Alfresco 的 Activiti 项目 (<http://activiti.org>) 是一个业务流程引擎。它支持 BPMN 2.0 的 XML 标准中的流程定义，其可以跨多个供应商的工具和 IDE 获得强大的支持。业务流程设计者使用工具（例如 WebMethods、Activiti 或 IBM）定义 BPMN 业务流程，然后

在 Activiti（或在任何其他工具）中运行定义的流程。Activiti 还提供建模环境，易于独立部署或在基于云的服务中使用，并且是 Apache 2 许可。Activiti 还为我们提供了便捷的 Spring Boot 自动配置功能。



在写作本书时，Activiti 项目的主要开发人员将该项目分出了一个新的 Apache 2 许可，完全向后兼容，称为 Flowable (<http://www.flowable.org/>)。如果你喜欢 Activiti，一定要留意这个空间。

在 Activiti 中，Process 定义了一个典型的流程。ProcessInstance 是给定进程的一次执行。ProcessInstance 由其流程变量唯一定义，这个变量参数化流程的执行。把它们想象成命令行参数，比如 Spring 批处理作业的 JobParameter 参数。

Activiti 引擎就像 Spring Batch 一样，将其执行状态保存在关系数据库中。如果你使用的是 Spring Boot 自动配置，它会自动为你安装相关的表，只要你的应用程序上下文中有 一个 DataSource 即可。示例 11-15 显示了相关的表格。

示例11-15 MySQL中的Activiti元数据表

```
mysql> show tables;
```

Tables_in_activiti
ACT_EVT_LOG
ACT_GE_BYTEARRAY
ACT_GE_PROPERTY
ACT_HI_ACTINST
ACT_HI_ATTACHMENT
ACT_HI_COMMENT
ACT_HI_DETAIL
ACT_HI_IDENTITYLINK
ACT_HI_PROCINST
ACT_HI_TASKINST
ACT_HI_VARINST
ACT_ID_GROUP
ACT_ID_INFO
ACT_ID_MEMBERSHIP
ACT_ID_USER
ACT_PROCDEF_INFO
ACT_RE_DEPLOYMENT
ACT_RE_MODEL
ACT_RE_PROCDEF
ACT_RU_EVENT_SUBSCR
ACT_RU_EXECUTION

```

| ACT_RU_IDENTITYLINK      |
| ACT_RU_JOB               |
| ACT_RU_TASK              |
| ACT_RU_VARIABLE          |
+-----+
24 rows in set (0.00 sec)

```

Spring Boot 自动配置要求在默认情况下，所有 BPMN 2.0 文档都位于应用程序的 `src/main/resources/processes` 目录中。在示例 11-16 中，给出了 BPMN 2.0 `signup` 流程的定义。

示例11-16 `signup.bpmn20.xml`业务流程定义

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:activiti="http://activiti.org/bpmn"
    xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
    typeLanguage="http://www.w3.org/2001/XMLSchema"
    expressionLanguage="http://www.w3.org/1999/XPath"
    targetNamespace="http://www.activiti.org/bpmn2.0">

    <process name="signup" id="signup">

        ①
        <startEvent id="start"/>

        ②
        <sequenceFlow sourceRef="start" targetRef="check-form"/>

        ③
        <serviceTask id="check-form" name="check-form"
            activiti:expression="#{checkForm.execute(execution)}"/>

        <sequenceFlow sourceRef="check-form"
            targetRef="form-completed-decision-gateway"/>

        ④
        <exclusiveGateway id="form-completed-decision-gateway"/>

        <sequenceFlow name="formOK" id="formOK"
            sourceRef="form-completed-decision-gateway"
            targetRef="send-confirmation-email">
            <conditionExpression xsi:type="tFormalExpression">${formOK == true}</conditionExpression>
        </sequenceFlow>
    </process>
</definitions>

```



```

<sequenceFlow id="formNotOK" name="formNotOK"
    sourceRef="form-completed-decision-gateway"
    targetRef="fix-errors">
    <conditionExpression xsi:type="tFormalExpression">${formOK == false}
    </conditionExpression>
</sequenceFlow>

5
<userTask name="fix-errors" id="fix-errors">
    <humanPerformer>
        <resourceAssignmentExpression>
            <formalExpression>customer</formalExpression>
        </resourceAssignmentExpression>
    </humanPerformer>
</userTask>

<sequenceFlow sourceRef="fix-errors" targetRef="check-form"/>

6
<serviceTask id="send-confirmation-email" name="send-confirmation-email"
    activiti:expression="#{sendConfirmationEmail.execute(execution)}"/>

<sequenceFlow sourceRef="send-confirmation-email"
    targetRef="confirm-email"/>

7
<userTask name="confirm-email" id="confirm-email">
    <humanPerformer>
        <resourceAssignmentExpression>
            <formalExpression>customer</formalExpression>
        </resourceAssignmentExpression>
    </humanPerformer>
</userTask>

<sequenceFlow sourceRef="confirm-email" targetRef="end"/>

<endEvent id="end"/>
</process>
</definitions>

```

- ❶ 第一个状态是 startEvent。所有流程都有一个定义好的 start 和 end。
- ❷ sequenceFlow 元素可以作为引擎的下一步走向。它们是合乎逻辑的，在工作流模型本身中被表示为线条。
- ❸ serviceTask 是流程中的一个状态。BPMN 2.0 定义使用 Activiti 的 activiti:expression

属性将处理委托给 Spring bean (称为 checkForm) 上的方法 execute(ActivityExecution)。Spring Boot 的自动配置激活了这种行为。

- ④ 表单被提交并检查后, checkForm 方法提供一个布尔型的流程变量, 称为 formOK, 它的值用于驱动下游的决策。流程变量是对给定流程中的参与者可见的上下文。流程变量可以是任何定义, 尽管我们倾向于尽可能仔细定义, 以便稍后在其他地方实际资源用来做声明检查。这个流程需要一个输入流程变量 customerId。
- ⑤ 如果表单无效, 则工作流向 fix-errors 用户任务。该任务被加入到工作清单, 并分配给一个人。此任务列表通过 Activiti 中的 TaskAPI 得到支持。查询任务列表并启动和完成任务。任务列表是为了模拟人类必须执行的工作。当流程到达这个状态时, 它暂停等待由一个人显式地完成。工作流从 fix-errors 任务返回到 checkForm 状态。如果表单现在已修复 (有效), 工作进行到下一个状态。
- ⑥ 如果表单是有效的, 那么工作流向 send-confirmation-email 服务任务。这只是代表另一个 Spring bean 来完成它的工作, 也许使用 SendGrid。
- ⑦ 该电子邮件应该包含一个链接, 当单击该链接时, 会触发一个 HTTP 端点, 然后完成未完成的任务, 并完成该流程。

这个流程十分简单, 但它提供了一个干净系统中可移动组件的典范。我们可以看到, 需要有东西来接受用户的输入、数据库中的客户记录和有效的 customerId, 我们可以使用 customerId 来检索下游组件中的记录。客户记录可能处于无效状态 (电子邮件可能无效), 可能需要用户重新访问。一旦事物处于正常工作状态, 状态就会转到发送确认电子邮件的步骤, 一旦收到并确认, 就会将流程转换到终止状态。

我们来看一个驱动这个流程的简单的 REST API。为了简洁起见, 我们没有推出 iPhone 客户端或 HTML5 客户端, 但这肯定是下一步。为了说明, REST API 设计得尽可能简单。下一步的计划可能是使用超媒体将客户端与 REST API 的交互从一个状态转换到另一个状态 (如示例 11-17 所示)。有关这种可能性的更多信息, 请参阅第 6 章中关于超媒体和 HATEOAS 的讨论。

示例11-17 驱动流程的SignupRestController

```
package com.example;

import org.activiti.engine.RuntimeService;
import org.activiti.engine.TaskService;
import org.activiti.engine.task.TaskInfo;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```



```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.util.Assert;
import org.springframework.web.bind.annotation.*;
```

```
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
```

```
@RestController
@RequestMapping("/customers")
class SignupRestController {
```

```
    public static final String CUSTOMER_ID_PV_KEY = "customerId";
```

```
    private final RuntimeService runtimeService;
```

```
    private final TaskService taskService;
```

```
    private final CustomerRepository customerRepository;
```

```
    private Log log = LogFactory.getLog(getClass());
```

```
    ①
    @Autowired
```

```
    public SignupRestController(RuntimeService runtimeService,
        TaskService taskService, CustomerRepository repository) {
        this.runtimeService = runtimeService;
        this.taskService = taskService;
        this.customerRepository = repository;
    }
```

```
    ②
    @PostMapping
```

```
    public ResponseEntity<?> startProcess(@RequestBody Customer customer) {
        Assert.notNull(customer);
        Customer save = this.customerRepository.save(new Customer(customer
            .getFirstName(), customer.getLastName(), customer.getEmail()));
```

```
        String processInstanceId = this.runtimeService.startProcessInstanceByKey(
            "signup",
            Collections.singletonMap(CUSTOMER_ID_PV_KEY, Long.toString(save.getId())))
            .getId();
        this.log.info("started sign-up. the processInstance ID is "
            + processInstanceId);
```

```
        return ResponseEntity.ok(save.getId());
```

```
}
```

③

```
@GetMapping("/{customerId}/signup/errors")
public List<String> readErrors(@PathVariable String customerId) {
    // @formatter:off
    return this.taskService
        .createTaskQuery()
        .active()
        .taskName("fix-errors")
        .includeProcessVariables()
        .processVariableValueEquals(CUSTOMER_ID_PV_KEY, customerId)
        .list()
        .stream()
        .map(TaskInfo::getId)
        .collect(Collectors.toList());
    // @formatter:on
}
```

④

```
@PostMapping("/{customerId}/signup/errors/{taskId}")
public void fixErrors(@PathVariable String customerId,
    @PathVariable String taskId, @RequestBody Customer fixedCustomer) {
```

```
    Customer customer = this.customerRepository.findOne(Long
        .parseLong(customerId));
    customer.setEmail(fixedCustomer.getEmail());
    customer.setFirstName(fixedCustomer.getFirstName());
    customer.setLastName(fixedCustomer.getLastName());
    this.customerRepository.save(customer);
```

```
    this.taskService.createTaskQuery().active().taskId(taskId)
        .includeProcessVariables()
        .processVariableValueEquals(CUSTOMER_ID_PV_KEY, customerId).list()
        .forEach(t -> {
            log.info("fixing customer# " + customerId + " for taskId " + taskId);
            taskService.complete(t.getId(), Collections.singletonMap("formOK", true));
        });
}
```

⑤

```
@PostMapping("/{customerId}/signup/confirmation")
public void confirm(@PathVariable String customerId) {
    this.taskService.createTaskQuery().active().taskName("confirm-email")
        .includeProcessVariables()
        .processVariableValueEquals(CUSTOMER_ID_PV_KEY, customerId).list()
        .forEach(t -> {
```



```

        log.info(t.toString());
        taskService.complete(t.getId());
    });
    this.log.info("confirmed email receipt for " + customerId);
}
}

```

- ❶ 控制器将使用一个简单的 Spring Data JPA 存储库以及由 Activiti Spring Boot 支持的自动配置的两个服务。RuntimeService 可以让我们询问流程引擎关于正在运行的流程。TaskService 可以让我们询问流程引擎正在运行的人工任务和任务列表。
- ❷ 第一个端点接受一个新 customer 记录并保存。新的客户将作为输入流程变量输入到一个新的流程中，其中客户的 customerId 被指定为流程变量。在这里，处理流向 check-form serviceTask，它将在名义上检查输入的电子邮件是否有效；如果有效，则发送确认电子邮件，否则，客户需要修改输入数据中的错误。
- ❸ 如果有任何错误，我们希望用户的 UI 阻止进度继续，于是进程排队用户任务。此端点查询此特定用户的任何未完成任务，然后返回未完成任务 ID 的集合。理想情况下，也可能返回验证信息，这些信息可以通过某种交互式体验来吸引用户的 UX。
- ❹ 一旦错误在客户端得到解决，更新的实体将被保存在数据库中，任务将被标记为完成。此时，流程进入 send-confirmation-email 状态，发送包含用户必须单击的链接的电子邮件以确认用户注册。
- ❺ 最后一步是 REST 端点查询给定客户的任何未完成的电子邮件确认任务。

我们以一个潜在的无效实体来启动这个流程，客户的状态需要确保是正确的，才能继续下去。我们对这个流程建模，以支持对实体的迭代，只要存在无效状态就回溯到适当的步骤。

我们通过包含两个 serviceTask 元素的 bean 的流程来完成检查：check-form 和 send-confirmation-email。

CheckForm 和 SendConfirmationEmail bean 没什么好说的，它们都是简单的 Spring bean。CheckForm 实现了一个简单的验证，确保名字和姓氏不为空，然后使用 Mashape 电子邮件验证 REST API（在 Spring 批处理中介绍过）来验证电子邮件是否有效。CheckForm bean 验证给定的 Customer 记录的状态，然后为正在运行的 ProcessInstance（一个名为 formOK 的布尔值）提供一个流程变量，之后决定是否继续该流程或强制用户再次尝试（如示例 11-18 所示）。

示例11-18 验证客户状态的CheckForm bean

```
package com.example;

import com.example.email.EmailValidationService;
import org.activiti.engine.RuntimeService;
import org.activiti.engine.impl.pvm.delegate.ActivityExecution;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Collections;
import java.util.Map;

import static org.apache.commons.lang3.StringUtils.isEmpty;

@Service
class CheckForm {

    private final RuntimeService runtimeService; ❶

    private final CustomerRepository customerRepository;

    private final EmailValidationService emailValidationService;

    @Autowired
    public CheckForm(EmailValidationService emailValidationService,
        RuntimeService runtimeService, CustomerRepository customerRepository) {
        this.runtimeService = runtimeService;
        this.customerRepository = customerRepository;
        this.emailValidationService = emailValidationService;
    }

    ❷
    public void execute(ActivityExecution e) throws Exception {
        Long customerId = Long.parseLong(e.getVariable("customerId", String.class));
        Map<String, Object> vars = Collections.singletonMap("formOK",
            validated(this.customerRepository.findOne(customerId)));
        this.runtimeService.setVariables(e.getId(), vars); ❸
    }

    private boolean validated(Customer customer) {
        return !isEmpty(customer.getFirstName()) && !isEmpty(customer.getLastName())
            && this.emailValidationService.isEmailValid(customer.getEmail());
    }
}
```


- ❶ 注入 Activiti RuntimeService。
- ❷ ActivityExecution 是一个上下文对象。你可以使用它来访问流程变量、流程引擎本身以及其他感兴趣的服务。
- ❸ 然后用它来表示测试的结果。

SendConfirmationEmail 具有相同的基本形式。

到目前为止，我们已经在单个节点的上下文中查看了 workflow。我们忽略了云最重要的特性：规模！对 workflow 进行建模有助于我们识别困难的并行处理。流程中的每个状态都需要输入（以流程变量和前提条件的形式），并提供输出（以副作用和流程变量的形式）。workflow 管理流程状态。

我们的业务逻辑根据状态在正确的时间执行。到目前为止，我们所做的一切都发生在同一个节点上，或者客户端正在使用的任何节点与系统进行交互，以满足用户分配的任务列表。如果自动处理花费了大量的时间，那么把这个工作转移到另一个节点——任何节点都是比较安全的！毕竟，这是云。我们有容量；这只是一个使用问题。

在 BPMN 流程中，serviceTask 元素是一个等待状态：引擎将暂停（并且处理任何持有的资源），直到它被显式地信号通知。任何外在事物都可以向这个流程发送信号：来自 Apache Kafka 代理的消息到达、电子邮件、按钮单击或长期运行流程结束。我们可以利用这种行为，并将可能长时间运行的处理移动到另一个节点，然后在完成时指示应该恢复流程。workflow 引擎会跟踪所有运行和正在运行的进程的状态，因此可以查询任何孤立的流程并相应地执行操作。workflow 使我们能够自然分解涉及的业务流程，将其分解为独立的步骤，并将其分布到集群中；它提供了工具，以便在工作分配中出现问题时能够恢复。

我们来看示例 11-19，了解如何将执行导出到 worker 节点的 Spring Cloud Stream 的 MessageChannel 定义中。正如我们在远程分区的 Spring Batch 作业中所做的那样，我们将根据 leader 节点和 worker 节点来思考解决方案。leader 节点就是发起处理的节点。leader 节点靠 HTTP 端点（http://localhost:8080/start）来启动处理。这个流程很简单，所以我们将把重点放在分配上。

示例 11-19 异步业务流程，async.bpmn20.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns:activiti="http://activiti.org/bpmn"
    id="definitions"
    xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
    typeLanguage="http://www.w3.org/2001/XMLSchema"
```

```

    expressionLanguage="http://www.w3.org/1999/XPath"
    targetNamespace="http://www.activiti.org/bpmn2.0">

<process id="asyncProcess">

    <startEvent id="start"/>

    <sequenceFlow id="f1" sourceRef="start" targetRef="spring-gateway"/>

    ❶
    <serviceTask id="spring-gateway"
        activiti:delegateExpression="#{gateway}"/>

    <sequenceFlow id="f2" sourceRef="spring-gateway"
        targetRef="confirm-movement"/>

    ❷
    <scriptTask id="confirm-movement" scriptFormat="groovy">
        <script>
            println 'Moving on..'
        </script>
    </scriptTask>

    <sequenceFlow id="f3" sourceRef="confirm-movement" targetRef="end"/>

    <endEvent id="end"/>

</process>

</definitions>

```

- ❶ 流程一开始就进入 `serviceTask` 状态。这里我们使用 Activiti 委托，一个名为 `gateway` 的 Spring bean 来处理这个状态。该流程将在该处理程序处停止，直到它接到信号。
- ❷ 接到信号后，流程将继续进行到 `scriptTask`，我们将看到在控制台上记录的 `Moving on!`。

`gateway` bean 是关键。Activiti `ReceiveTaskActivityBehavior` 是一个将消息写入 Spring Cloud Stream 支持的 `leaderRequests MessageChannel` 的实现。`executionId` 是消息的有效载荷，并且稍后需要用信号发送请求。我们必须注意在消息标题或有效载荷中使其持久化（就像我们在这个例子中所做的那样）。当在另一个 Spring Cloud Stream 支持的 `MessageChannel` 上接收到回复消息时，`respondFlow` 会调用 `RuntimeService#signal(executionId)`，进而恢复执行流程（如示例 11-20 所示）。

示例11-20 LeaderConfiguration定义请求发送到的方式，以及worker节点在回复时发生的情况

```
package com.example;
```

```
import org.activiti.engine.ProcessEngine;
import org.activiti.engine.impl.bpmn.behavior.ReceiveTaskActivityBehavior;
import org.activiti.engine.impl.pvm.delegate.ActivityBehavior;
import org.activiti.engine.impl.pvm.delegate.ActivityExecution;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;
```

```
@Configuration
```

```
@Profile(Profiles.LEADER)
```

```
class LeaderConfiguration {
```

❶

```
@Bean
```

```
ActivityBehavior gateway(LeaderChannels channels) {
```

```
    return new ReceiveTaskActivityBehavior() {
```

```
        @Override
```

```
        public void execute(ActivityExecution execution) throws Exception {
```

```
            Message<?> executionMessage = MessageBuilder.withPayload(execution.getId())
                .build();
```

```
            channels.leaderRequests().send(executionMessage);
```

```
        }
```

```
    };
```

```
}
```

❷

```
@Bean
```

```
IntegrationFlow repliesFlow(LeaderChannels channels, ProcessEngine engine) {
```

```
    return IntegrationFlows.from(channels.leaderReplies())
```

```
        .handle(String.class, (executionId, map) -> {
```

```
            engine.getRuntimeService().signal(executionId);
```

```
            return null;
```

```
        }).get();
```

```
    }
```

```
}
```

- ❶ 流程一开始就进入 `serviceTask` 状态。这里我们使用 Activiti 委托，一个名为 `gateway` 的 Spring bean 来处理这个状态。该流程将在该处理程序处停止，直到它接到信号。
- ❷ 接到信号后，流程将继续进行到 `scriptTask`，我们将在控制台上看到 `Moving on!`。

worker 节点只能使用 `MessageChannel` 实例：Activiti 或 workflow 引擎没有意识（超出 `String executionId` header）。worker 节点可以自由地做自己想做的事情，也许它们会运行一个 Spring Batch Job 或者一个 Spring Cloud Task。工作流适用于任何长时间运行的流程，因此无论你想要做什么都可以在这里完成：视频代码转换、文档生成、图像分析等（如示例 11-21 所示）。

示例 11-21 WorkerConfiguration 定义了请求如何发送到 work 节点，以及 worker 节点回复的情况

```
package com.example;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.support.GenericHandler;

@Configuration
@Profile({Profiles.WORKER})
class WorkerConfiguration {

    @Bean
    IntegrationFlow requestsFlow(WorkerChannels channels) {

        Log log = LogFactory.getLog(getClass());

        ❶
        return IntegrationFlows.from(channels.workerRequests())
            .handle((GenericHandler<String>) (executionId, headers) -> {
                ❷
                headers.entrySet().forEach(e -> log.info(e.getKey() + '=' + e.getValue()));
                log.info("sending executionId (" + executionId + ") to workerReplies.");
                return executionId;
            }).channel(channels.workerReplies()) ❸
            .get();
    }
}
```


- ① 每当有新消息到达时
- ② 枚举标题，并处理我们想要处理的任何其他业务逻辑，确保持久化 `executionId` 有效载荷。
- ③ 通过 `workerReplies` 通道发送，返回 `leader` 节点。

使用消息传递的分布式

尽管本章讨论了很多长时间运行的流程（Spring Batch、Spring Cloud Task 和 Workflow），但是消息传递才为在云原生的书中讨论该主题赋予了真正的价值。我们在利用云的规模赋予我们的能力时，这些“陈旧”的东西又成为了我们的得力工具。

总结

本章我们所学习的仅仅是数据处理可行性的一些皮毛而已。当然，这些技术中的每一种技术都包含许多不同的技术。例如，我们可以将 Apache Spark 或 Apache Hadoop 与 Spring Cloud Data Flow 结合使用。它们组合起来很好用。使用 Spring Cloud Stream 作为消息传递结构来扩展跨集群的 Spring Batch Job 处理仅仅是冰山一角。

第 12 章

数据集成

微服务针对团队的优化易于系统的独立演进。每个团队与组织的其他部门互相独立，团队独立完成自己的可交付成果或功能——独立的代码库、单独的发布周期以及可能独立的技术！这种隔离的作用是，服务是分布式的。服务边界是明确的，数据访问通过服务边界发生。这意味着进程的分布式和网络分区。在数据管理章节中，我们研究了如何为有界的上下文建模，并与 MongoDB 或 Redis 等流行的数据源交互。创建管理自己数据源的个别服务很简单，问题是这些节点之间如何通信？如何保证它们的状态达成一致？

在本章中，我们将介绍几种不同的方式，从不同的微服务中获取并整合数据。我们试图解决的关键问题之一是分布式环境下的数据完整性问题。分布式系统的字面意思是广泛和全面的。还有一些开创性的论文，例如“Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System” (<http://bit.ly/2vjbzaN>)（管理 Bayou 中的更新冲突，弱连接的复制存储系统），打破了分布式数据库体系结构一致性的问题。还有 Eric Brewer 的一篇论文，“CAP Theorem” (<http://dl.acm.org/citation.cfm?id=343502>)（CAP 定理），该文指出，任何分布式系统只能够满足三个性质中的两个：

- 一致性（Consistency），相当于拥有一个最新的数据副本。
- 数据的高可用性（Availability）（用于更新）。
- 网络分区（Partition）的容忍性。

在实践中，CAP 通常为了保证分区容错性而牺牲一部分的高可用性和一致性。然而，只有很少的情况需要完美的高可用性和一致性。相反，我们要研究各种能够使系统达到最终一致状态以及整合管理失败的模式或补偿性措施。本章并非旨在强调和解决所有可能的故障模式！系统失败可能会有很多种方式，正如 Kyle Kingsbury 的史诗般的分布式系统类所示 (<https://github.com/aphyr/distsys-class>)！

我们平时工作中接触的数据集的性质，已经从仅适合我们自己的应用程序、传统的面向

工作日、离线的批处理和有限工作负载开始向国际、全天候、始终在线以及无限的基于事件和流的工作负载的方向转变。无论我们选择使用哪种技术，都必须支持这些不同类型的工作负载。

分布式事务

乍一看，分布式事务似乎可以保证服务之间的一致性。在 Java 中通过 JTA API 来支持分布式事务。JTA 是 X/Open XA 架构的一个实现。在两阶段提交事务中，有一个协调器用于在事务中谋取资源。然后告诉每个资源准备提交。资源回应说它们已经准备好了，然后最终被要求同时提交。如果每个资源都回复它们已经提交，那么事务成功。如果不是，则资源被要求回滚。

分布式事务在云原生世界中是一个糟糕的选择。协调器是一个单点故障——它需要在每个节点之间进行大量的通信，才能有效地处理事务。这种通信使网络产生不必要的过载。分布式事务要求所有参与的资源都知道事务协调器，而这个协调器不可能存在于没有使用 X/Open 协议的 REST API 的系统中。有时候这种方法是必需的，尽管我们不推荐使用。如果你想了解如何提升和转移现有的基于 JTA 的应用程序，请参阅附录 A 中关于分布式事务 JTA 的讨论。

故障隔离和优雅的降级

我们的最终目标是从分布式服务中获得正确的结果。即使服务失败，我们也可以通过多次重试请求来弥补，直到服务可用。换句话说就是坚持！我们可以使用 Spring Retry 库重试请求，或者是任何潜在的不稳定工作。Spring Retry 是从 Spring Batch 中提取出来的（我们在第 11 章中讨论了 Spring Batch 和其他支持长时间运行的技术），现在可以独立于 Spring Batch 使用。你可以使用 Spring Retry 来尝试建立像数据源或消息队列这样的中间件连接——这是一种当以不确定的顺序建立基础架构时特别有用的模式，例如，数据库在依赖它的服务已经启动后才开始服务外部请求。你可以使用 Spring Retry 来调用状态未知的下游服务。

我们来看一个简单的调用另一个服务的客户端。如果调用成功返回服务响应；如果调用失败，则抛出一个异常，Spring Retry 将会路由到一个使用 `@Recover` 注解的处理方法。恢复方法返回与可恢复方法相同的响应。在我们的例子中，会返回字符串 `OHAI`。它会尽可能多地重试。默认情况下会调用三次。每次重试之前都会增加停止的时间，最终失败（如示例 12-1 所示）。在配置类上使用 `@EnableRetry` 注解来激活 Spring Retry。

示例12-1 Spring Retry帮助我们重试，每次尝试都会增加退避周期，以调用潜在的下游服务

```

package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.retry.annotation.Backoff;
import org.springframework.retry.annotation.Recover;
import org.springframework.retry.annotation.Retryable;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.util.Date;
import java.util.Map;

@Component
public class RetryableGreetingClient implements GreetingClient {

    private final RestTemplate restTemplate;

    private final String serviceUri;

    private Log log = LogFactory.getLog(getClass());

    @Autowired
    public RetryableGreetingClient(RestTemplate restTemplate,
        @Value("${greeting-service.uri}") String domain) {
        this.restTemplate = restTemplate;
        this.serviceUri = domain;
    }

    ❶
    @Retryable(include = Exception.class, maxAttempts = 4,
        backoff = @Backoff(multiplier = 5))
    @Override
    public String greet(String name) {
        long time = System.currentTimeMillis();

        Date now = new Date(time);

        this.log.info("attempting to call the greeting-service " + time + "/"
            + now.toString());

        ParameterizedTypeReference<Map<String, String>> ptr =

```



```

new ParameterizedTypeReference<Map<String, String>>() {
};

return this.restTemplate
    .exchange(this.serviceUri + "/hi/" + name, HttpMethod.GET, null, ptr, name)
    .getBody().get("greeting");
}

②
@Recover
public String recoverForGreeting(Exception e) {
    return "OHAI";
}
}

```

- ❶ 这是可能失败的方法，如果这样做会引发异常。如果我们想要在特定类型的失败中重试和恢复，可以指定一个特殊类型的 `Exception`。
- ❷ 你可以指定尽可能多的恢复方法，根据 `Exception` 的类型执行不同的恢复方法。

Spring Retry 可能正是快速简单的恢复和智能退避所需要的。然而，有时服务可能需要更多时间才能恢复，而大量请求和重试可能会对网络产生负面影响。我们可以使用断路器，根据之前的请求是否成功来控制请求，或者如同使用 Spring Retry 一样回退到恢复方法。

断路器实现了一些与 Spring Retry 相同的东西。这是一个在特殊情况下调用回退机制的组件。不过，它不仅仅是 try-catch 处理程序。它保持一个状态机。如果断路器观察到足够多的连续故障，则会直接将请求路由到回退路径，前面的尝试调用已知故障路径。这与 Spring Retry 不同，因为 Spring Retry 不会主动转移请求。

目前，Spring 支持两种断路器。Spring Retry 项目还提供了一个断路器，我们可以使用 Netflix Hystrix 断路器的 Spring Cloud 集成。我们来看看 Netflix Hystrix 是怎么回事。

什么时候应该在 Spring Retry 上使用 Hystrix？目前，Hystrix 有更好的观察性。还有一个附带的 Hystrix 仪表盘，它把输入服务器发送的事件流作为输入，从而来显示通过断路器的请求流。Hystrix 的 Spring 集成（由 Javanica 社区库提供）可能会有点不合时宜，因为它需要使用字符串来命名方法，Spring Retry 的方法依赖于异常和异常处理程序。Hystrix 有能力使用单独的线程池来隔离请求，而 Spring Retry 目前还没有这个能力。如果对服务依赖的请求运行时间过长，则线程隔离的 Hystrix 断路器可以更容易地从失败的调用中退出。当然，这需要更多的资源，因为每个客户端都有自己的线程池。Hystrix 还支持基于信号量的隔离，这有些像使用 Spring Retry 的效果。如果你可以使用基于信

号量的隔离，有限的计算资源，想要一个更简洁的 API（还保持连续性），请使用 Spring Retry。如果必须得有基于线程的隔离或希望断路器可视化，请使用 Hystrix。

在示例 12-2 中，给出了一个正在调用包含潜在不稳定因素的服务客户端。

示例12-2 这个断路器给我们的下游服务以喘息时间

```
package demo;
```

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;
```

```
import java.util.Date;
```

```
import java.util.Map;
```

```
@Component
```

```
public class CircuitBreakerGreetingClient implements GreetingClient {
```

```
    private final RestTemplate restTemplate;
```

```
    private final String serviceUri;
```

```
    private Log log = LogFactory.getLog(getClass());
```

```
    @Autowired
```

```
    public CircuitBreakerGreetingClient(RestTemplate restTemplate,
```

```
        @Value("${greeting-service.uri}") String uri) {
```

```
        this.restTemplate = restTemplate;
```

```
        this.serviceUri = uri;
```

```
    }
```

```
    ❶
    @HystrixCommand(fallbackMethod = "fallback")
    public String greet(String name) {
```

```
        long time = System.currentTimeMillis();
```

```
        Date now = new Date(time);
```

```
        this.log.info("attempting to call " + "the greeting-service " + time + "/"
            + now.toString());
```

```
        //@formatter:off
```



```

ParameterizedTypeReference<Map<String, String>> ptr =
    new ParameterizedTypeReference<Map<String, String>>() {
    };
//@formatter:on
return this.restTemplate
    .exchange(this.serviceUri + "/hi/" + name, HttpMethod.GET, null, ptr, name)
    .getBody().get("greeting");
}

public String fallback(String name) {
    return "OHAI";
}

}

```

❶ 该方法可能失败，如果失败，则抛出一个异常。

断路器可以是开路的也可以是闭路的。如果是闭合的，那么断路器将尝试调用快乐路径，在遇到异常的时候调用回退方法。如果路线断开，断路器会直接将请求路由到回退方法。但最后，路线将再次关闭并试图重新引入流量。如果再次失败，它会重新打开。断路器还会发出服务器发送的事件流，我们可以使用 Hystrix Dashboard 和 Spring Cloud Turbine 进行监控。有关详细信息，请参阅第 13 章中的讨论。

如果你正在使用 Cloud Foundry 这样的平台，那么没有必要让下游服务关闭很长时间。Cloud Foundry 有一个心跳机制，其可以自动重新启动一个关闭的服务，甚至可以根据需求自动扩展服务。重试和断路器使得客户体验在服务失败不可避免的情况下容易降级。Netflix 和其他高效能组织将实现优雅的降级。想象一下，假如搜索引擎服务调用失败，如果客户端显示堆栈跟踪，将会多么影响用户体验。

这些例子中的恢复方法是前期考虑到的。我们必须事先考虑失败案例和在失败的情况下如何做出补偿。这是一种美德。假设错误必然发生，然后面对各种可能的错误在前期构建出更加健壮的系统。我们已经研究了在没有其他资源辅助的情况下，如何增加服务失败时的弹性。如果你了解借助其他资源带来的副作用，就会发现这样做的效果很好。

我们来看一下使用消息传递作为保证服务最终在一致状态下收敛的方式，即使服务在这个时候停止。使用消息传递方式构建的架构已经隐含了故障将随时发生的事实，并对其做出了相应的架构设计，而断路器试图反应和弥补分区之间不可避免的故障问题。解释消息传递和应用程序集成的权威著作 *Enterprise Integration Patterns*（企业集成模式）（Addison-Wesley）的合著者 Gregor Hohpe 说：“消息传递是一个更诚实的架构。它承认解耦的服务有时可能会发生故障。”我们在第 10 章讨论消息传递时，学习了 Spring 生态系统对于基本消息传递的支持。请参考该章的内容。下面我们学习消息传递的有关技术。

saga 模式

在一个典型的分布式系统中，工作将会跨越多个节点。一个事实是，一些请求将失败，并产生不一致的结果。saga 模式最初设计用于在同一个节点上处理长期事务。传统的长期事务是分布式系统中的瓶颈，因为资源必须在事务处理期间保持不变。这些资源对系统是不可用的。这限制了系统的吞吐量。

Hector Garcia-Molina 在 1987 年提出了 saga 模式。saga 是可以写成子事务序列的长期事务。它是子事务的集合，在分布式系统中我们称其为请求 (*request*)。事务需要是可交错的，它们不能相互依赖，必须可以重新排序。每个事务还必须定义补偿性事务，以消除事务的影响，使系统回到语义一致的状态。这些补偿性事务是由开发人员定义的。在设计系统时需要有一些预先考虑，以确保系统始终处于语义一致的状态。saga 模式权衡了系统的可用性和一致性；在整个 saga 完成之前，每个子事务的作用对系统的其他部分是可见的。

saga 基于 saga 执行协调器 (SEC) 的概念。saga 执行协调器是 saga 日志的守护者。saga 日志用来记录正在进行的事务并持续记录事务的进度。然而，saga 执行协调器在这个过程中并没有自己的状态。如果出现故障，那么搞一个新的执行协调器来取代它就很容易了。SEC 跟踪 saga 中有哪些交易正在进行中，以及它们已经取得的进展。如果交易失败，SEC 必须开始补偿事务。如果补偿事务失败，SEC 将会重试（再次重试）。这对我们的架构有一些影响。saga 事务应该被设计为最多一次 (*at-most-once*) 语义。补偿 saga 事务应该被设计为至少一次 (*at-least-once*) 语义；它们应该是幂等的，如果多次执行，则不会留下明显的副作用。关于这个特殊模式有很多很好的资源，但是这里衷心推荐 Caitie McCaffrey 在 GOTO 2015 关于 saga 模式的演讲 (<https://www.youtube.com/watch?v=xDuwrtwYHu8>)。在 Activiti 提供的工作流系统基础之上建立一个 SEC 并不难。有关 Activiti 和工作流的更多信息，请参考第 11 章的有关内容。

CQRS（命令查询责任分离）

系统中的每个微服务都是有界上下文而言的，它们具有内部一致性。有界环境中的实体意味着一件事，而且只有一件事，并且只存在于一个特定的有界环境中。每个有界的上下文都有自己的数据库，有多种方式来利用公共数据。服务可以管理写入它们自己数据库中的数据，但它们可能需要只读访问其他系统中的数据。

假设你正在构建电子商务引擎，可能会有如下服务：管理下单的服务、管理在线购物车的服务、管理发货的服务、客户服务、商品目录服务及商品搜索引擎等。虽然所有这些服务的最终目标都是将商品交给客户，但它们的领域模型并不相同。发货服务的领域模

型可能涉及包裹和运载工具有关的各种细节，这些细节将会阻碍购物车服务的领域模型构建。

在这样的系统中，商品目录将管理与商品相关的所有信息的更新。其他系统，如商品搜索引擎，可能会管理一个搜索索引，将用户对商品的评论和商品描述等内容存储在 Elasticsearch 集群中。它不会管理商品的信息，如价格、某个商品位于哪间仓库等信息。

微服务可能包含一个或多个聚合。聚合是根对象，对其子项的操作都是原子的：订单中的每一项商品、客户支持查询及进度跟踪、乘客及其行李、病人及其病历等。

如何在既不需要共享数据库，还不违反我们最初期望的微服务封装的情况下，确保写入的微服务（写入事务性地更新聚合）对于其他服务是可见的？是否需要在数据写入时更新所有其他以某种方式依赖于这些数据的服务？（这不会只是减慢写入路径吧？）另外，如何支持不同服务中共享数据的读取？毕竟，理想的商品搜索引擎最好由全文搜索引擎来提供，而不是 RDBMS，其中的数据没有真正的表示！

CQRS（命令查询责任分离）是由 Greg Young 创造的模式，它提供了一个解决方案：将读取与写入分开。命令消息驱动服务中聚合的更新。如果你想问系统问题，只需要发送查询。通过视图（或查询模型）支持查询。查询返回结果，但没有任何可观察到的副作用。

无论何时只要有命令更新领域模型并被持久化，就会有事件（消息）触发所有的观察查询模型来更新它们自己的表示。如果哪个查询模型最终是它处理过的所有事件的总和，那就意味着可以从头开始重放事件来重建视图模型。事实上，如果你保存了事件存储中的每一个事件，那么你就可以重建整个系统状态！这被称为事件溯源。我们不需要使用 CQRS 中的事件溯源，但是一个服务于另一个。获得首个事件存储切入点并不难。很多人使用 Apache Kafka 作为事件存储，还有人使用 SQL 数据库。甚至还有专门的事件存储，如 Greg Young 的 Event Store (<https://geteventstore.com>) 和 Chris Richardson 的 Eventuate (<http://eventuate.io/>)，这些都是很好的选择。

下面我们通过一个简单的 REST API 来管理投诉单。愤怒的顾客想要投诉。

写入命令服务（可能通过客户端与 REST API 通信），然后在命令总线上发布命令。一个命令可能代表有新投诉提交、投诉的评论的增加，或者投诉的客户支持代表的增加。调用命令处理程序来处理该命令，并首先验证传入的命令。如果一切正常，它会发布一个或多个域事件。

事件处理程序可以以很多方式响应新的域事件。领域事件的主要结果是某种聚合状态的变化。一个域事件也可以产生更多的事件，并发送给其他微服务。这就是为什么许多微服务的开发者被 CQRS 吸引的原因——CQRS 是一种发布和订阅源自有界上下文以外的

应用程序域事件的方式。这种方式为我们提供了一种用来确保域数据的参照完整性的机制。发送到其他微服务的消息可以用来处理域事件，这些域事件允许维护与分布式系统中其他记录的域数据相关的恼人的外键关系。查询服务随后响应查询以获得处理后的状态，并从查询模型数据存储中返回“未更改”。

默认情况下，域（命令）模型和查询模型存在于单个应用程序的同一个进程中。当 CQRS 与微服务相结合时，情况会变得有点复杂。我们来看一个实现 CQRS 的“简单”微服务。



CQRS 系统是分布式的单体吗？大多数人在想到一个单一的微服务时，他们首先想到的是一个独立的服务组件。在大多数情况下，微服务是为专注于做好一件事情的应用程序而构建的。最重要的是，该应用可以独立于其他服务来升级和部署。传统的 CQRS 系统有多个组件，有人会问：“这是一个分布式的单体吗？”当一个功能被交付时需要同时部署多个单独的组件，分布式单体就产生了。微服务赋予了小型独立团队不断为更大的微服务生态系统提供新的功能的能力。由于 CQRS 的独立组件仍然可以独立部署，因此我们可以说每个部署单元仍然满足独立交付功能投入生产的最低要求。微服务的一个功能应该最多只需要一个可部署单元。

虽然我们可以直接使用 Spring Cloud Stream 和 Spring Data 来实现 CQRS 架构，但是我们必须填补一些空白，比如事件溯源和命令路由。所以我们将使用 Axon Framework(<http://axonframework.org>)。Axon 是由 Allard Buijze (<https://twitter.com/allardbz>) 及其在 Trifork 的团队开发的框架。Axon 有很多好东西：它已经存在了 7 年（并且还在继续开发），而且它一直都可以和 Spring 一起很好地运行。它甚至提供了一个方便的 Spring Boot 自动配置功能，作为推荐的方法提供。Axon 一直在不断发展，以便更加融入 Spring Cloud 生态系统。



Axon 项目以轴突终端得名。在神经系统中，轴突（神经元的一个组成部分）传输电信号。这些神经元复杂地连接在一起。轴突终端负责将这些信号从一个神经元传递到另一个神经元。

我们来看一个 Axon 应用程序的例子。

投诉 API

让我们来实现假设的“投诉”服务。另外，我们还将构建一个支持投诉报告统计的微服务。主要的投诉服务将支持创建和结束投诉，并为投诉添加评论。我们希望确保评论不能被添加到不再公开的投诉中。对于这些操作——添加投诉，关闭它并添加注释，所有的写

操作都使用命令进行通信。所有的命令都是消息也是对象——具有命令处理程序将用来执行其工作的参数。命令消息几乎没有行为——只有数据。



在这个例子中，以及本书的其他地方，将大量使用 Lombok 项目。Lombok 是一个支持编译时注解处理的框架。它定义了几个有用的编译时注解，如 `@Data`（它为对象中的字段添加了 `getter` 和 `setter`，并提供了有用的 `toString` 和 `equals` 方法）和 `@AllArgsConstructor` 或 `@NoArgsConstructor`（为类中的所有字段生成构造函数，以及都不接受参数的构造函数）。

投诉 API 的入口点是一个 REST API。HTTP 请求在 Spring MVC 处理程序方法中被处理，然后在命令总线上调度命令（如示例 12-3 所示）。

示例12-3 投诉API

```
package complaints;

import org.axonframework.commandhandling.gateway.CommandGateway;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.util.UriComponents;
import org.springframework.web.util.UriComponentsBuilder;

import java.net.URI;
import java.util.*;
import java.util.concurrent.CompletableFuture;

import static org.springframework.http.MediaType.APPLICATION_JSON_VALUE;

//@formatter:off
@RestController
@RequestMapping(value = "/complaints",
    consumes = APPLICATION_JSON_VALUE,
    produces = APPLICATION_JSON_VALUE)
class ComplaintsRestController {
    //@formatter:on

    private final CommandGateway cg;

    @Autowired
    ComplaintsRestController(CommandGateway cg) {
        this.cg = cg;
    }
}
```



❶

```
@PostMapping
CompletableFuture<ResponseEntity<?>> createComplaint(
    @RequestBody Map<String, String> body) {

    String id = UUID.randomUUID().toString();
    FileComplaintCommand complaint = new FileComplaintCommand(id,
        body.get("company"), body.get("description"));

    return this.cg.send(complaint).thenApply(
        complaintId -> {
            URI uri = uri("/complaints/{id}",
                Collections.singletonMap("id", complaint.getId()));
            return ResponseEntity.created(uri).build();
        });
}
```

❷

```
@PostMapping("/{complaintId}/comments")
@ResponseStatus(HttpStatus.NOT_FOUND)
CompletableFuture<ResponseEntity<?>> addComment(
    @PathVariable String complaintId, @RequestBody Map<String, Object> body) {

    Long when = Long.class.cast(body.getOrDefault("when",
        System.currentTimeMillis()));

    AddCommentCommand command = new AddCommentCommand(complaintId, UUID
        .randomUUID().toString(), String.class.cast(body.get("comment")),
        String.class.cast(body.get("user")), new Date(when));

    return this.cg.send(command).thenApply(commentId -> {

        Map<String, String> parms = new HashMap<>();
        parms.put("complaintId", complaintId);
        parms.put("commentId", command.getCommentId());

        URI uri = uri("/complaints/{complaintId}/comments/{commentId}", parms);

        return ResponseEntity.created(uri).build();
    });
}

@DeleteMapping("/{complaintId}")
CompletableFuture<ResponseEntity<?>> closeComplaint(
    @PathVariable String complaintId) {
    CloseComplaintCommand csc = new CloseComplaintCommand(complaintId);
    return this.cg.send(csc).thenApply(none -> ResponseEntity.notFound().build());
}
```




```

}

private static URI uri(String uri, Map<String, String> template) {
    UriComponents uriComponents = UriComponentsBuilder.newInstance().path(uri)
        .build().expand(template);
    return uriComponents.toUri();
}
}

```

- ❶ 这些写操作（如 HTTP POST 请求）经过 /complaints 端点，然后使用 CommandGateway 将命令发送到 CommandBus。CommandGateway 上 #send 的结果是 CompletableFuture <T>。Spring MVC 知道如何使用 CompletableFuture <T> —— 只有当 CompletableFuture 的结果已经实现时，才会向客户端返回一个响应。同时，它会解锁 HTTP 请求处理线程，将工作移到后台线程。
- ❷ 为了给指定的投诉添加评论，AddCommentCommand 必须在命令中携带父投诉的标识符。这将命令与聚合的正确实例绑定。

命令网关根据类型异步地将命令分派到适当的命令处理程序。最终，命令网关委托给一个命令路由器，该命令路由器解析了应该调用命令处理程序的位置。默认情况下它在同一个节点上，但是有一些替代的策略来对这些调用做负载均衡。一个实现支持 JGroups，另一个实现支持 Spring Cloud DiscoveryClient 服务位置抽象。

使用 Axon 的说法，命令处理程序仅仅是一个使用 @CommandHandler 注解的 bean 中的一个方法。任何 Spring bean 都可以定义命令处理程序。我们的 API 支持三种操作，所以至少有三个命令：创建投诉（FileComplaintCommand）、添加评论（AddCommentCommand）、关闭投诉（CloseComplaintCommand），如示例 12-4、示例 12-5、示例 12-6 所示。

示例12-4 FileComplaintCommand命令表示应该创建一个新的投诉

```

package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.axonframework.commandhandling.TargetAggregateIdentifier;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class FileComplaintCommand {

    @TargetAggregateIdentifier
    private String id; ❶
}

```



```
private String company, description; ❷

}
```

- ❶ 稍后会再次讨论 @TargetAggregateIdentifier 的意义。
- ❷ 这是一个普通的 Java 对象——一个 POJO。我们需要了解投诉本身，以及我们希望提出投诉的公司。

示例12-5 AddCommentCommand命令表示应将评论添加到现有的投诉中

```
package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.axonframework.commandhandling.TargetAggregateIdentifier;

import java.util.Date;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class AddCommentCommand {

    @TargetAggregateIdentifier
    private String complaintId; ❶

    private String commentId, comment, user;

    private Date when;
}
```

- ❶ 尽管最终在我们的查询模型中，这将导致它在子关系的一个或者多个对象中出现，但它仍然是根据此处的 ID 来管理投诉的。

示例12-6 CloseComplaintCommand命令表示投诉应该被关闭

```
package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.axonframework.commandhandling.TargetAggregateIdentifier;
```

```
❶
@Data
```




```

@AllArgsConstructor
@NoArgsConstructor
public class CloseComplaintCommand {

    @TargetAggregateIdentifier
    private String complaintId;
}

```

❶ 我们唯一需要知道的事情是在哪个实体上执行。

命令处理程序的任务是验证命令对象有效负载，检查命令是否有效，然后操作域模型的状态。命令处理程序可以加载一个 `JdbcTemplate` 或一个 `Spring` 数据库，并在那里执行写入。但是它需要发布一个事件，以便感兴趣的观察者可以相应地更新自己的查询模型。在这里，命令处理程序可以将 `Axon` 事件总线上的事件分派给所有感兴趣的监听者。所有组件也可以通过指定事件处理程序来表达对事件的兴趣。建议不要在命令处理程序中执行写入操作，而是处理程序发布事件，然后在事件处理程序中执行写入操作。通过这种方式来训练自己就像是在更新第三方 API 一样来更新自己的域。

命令之间可能需要一些协调。如果无法加载和检查父项申诉，那么在创建子注释时，我们怎们知道 `AddCommentCommand` 的命令处理程序中的投诉是否仍处于打开状态呢？无论怎样，我们都需要这些信息。正常的 `Spring` 组件不会这样做。一个名义上的 `Spring` 组件应该是一个无状态的单例。对常规 `Spring` 组件的每个事件处理函数的调用都被路由到一个无状态的单例 bean，这个 bean 不应该记录与投诉有关的先前事务。

再来看下 `Axon` 聚合。聚合是一个有状态的组件，其标识符是在处理域事件之后应用突变的结果。`Axon` 使用 `Axon` 存储库实现来加载聚合。有些存储库本身存储聚合（例如使用 `JPA`），而其他存储则使用过去的事件重建当前状态（这被称为事件溯源）。可以根据我们的系统要求做出选择。我们的是一个事件溯源聚合：它将重放事件以重新创建任何给定聚合的最新状态。我们来看一个例子，然后分析发生了什么（见示例 12-7）。

示例12-7 `ComplaintAggregate`聚合既包含命令处理程序又包含事件源处理程序

```

package complaints.command;

import complaints.*;
import org.axonframework.commandhandling.CommandHandler;
import org.axonframework.commandhandling.model.AggregateIdentifier;
import org.axonframework.eventsourcing.EventSourcingHandler;
import org.axonframework.spring.stereotype.Aggregate;
import org.springframework.util.Assert;

import static org.axonframework.commandhandling.model.AggregateLifecycle.apply;

```



1

```
@Aggregate
public class ComplaintAggregate {
```

2

```
@AggregateIdentifier
private String complaintId;

private boolean closed;
```

```
public ComplaintAggregate() {
}
```

3

```
@CommandHandler
public ComplaintAggregate(FileComplaintCommand c) {
    Assert.hasLength(c.getCompany());
    Assert.hasLength(c.getDescription());
    apply(new ComplaintFiledEvent(c.getId(), c.getCompany(), c.getDescription()));
}
```

4

```
@CommandHandler
public void resolveComplaint(CloseComplaintCommand ccc) {
    if (!this.closed) {
        apply(new ComplaintClosedEvent(this.complaintId));
    }
}
```

5

```
@CommandHandler
public void addComment(AddCommentCommand c) {
    Assert.hasLength(c.getComment());
    Assert.hasLength(c.getCommentId());
    Assert.hasLength(c.getComplaintId());
    Assert.hasLength(c.getUser());
    Assert.notNull(c.getWhen());
    Assert.isTrue(!this.closed);
    apply(new CommentAddedEvent(c.getComplaintId(), c.getCommentId(),
        c.getComment(), c.getUser(), c.getWhen()));
}
```

6

```
@EventSourcingHandler
protected void on(ComplaintFiledEvent cfe) {
    this.complaintId = cfe.getId();
    this.closed = false;
```




```

    }

    ⑦
    @EventSourcingHandler
    protected void on(ComplaintClosedEvent cce) {
        this.closed = true;
    }
}

```

- ❶ @Aggregate 构造型注解表明这个对象是 Spring 生命周期管理的 bean。
- ❷ 当在 CommandGateway 上发送一个命令时，它的 @TargetAggregate 属性通过其 @AggregateIdentifier 注解字段中的一个值链接到该聚合。
- ❸ 在分派 FileComplaintCommand 时，将创建一个新的聚合实例。如果验证成功，则分派 ComplaintFiledEvent。ComplaintFiledEvent 被写入事件存储，然后分派给这个聚合和其他组件中的任何事件处理程序和事件溯源处理程序（如 on(ComplaintFiledEvent cce)）
- ❹ 当 CloseComplaintCommand 被分派时，投诉将被关闭。如果验证成功，则发送 ComplaintClosedEvent。ComplaintClosedEvent 被写入事件存储，然后分派给该聚合和其他组件中的任何事件处理程序和事件溯源处理程序（例如 on(ComplaintClosedEvent cce)）。
- ❺ 当发送 AddCommentCommand 时，新的评论将被添加到投诉中。如果验证成功，则分派 CommentAddedEvent。
- ❻ 第一个事件溯源处理程序初始化一个聚合，为其提供一个聚合标识符，并（冗余地）将 closed 布尔值初始化为 false。
- ❼ 第二个事件溯源处理程序在接收到 ComplaintClosedEvent 时将 closed 布尔值设置为 true。

因此，Axon 支持两种类型的事件监听器。如果事件要在聚合之外进行处理，那么在指定事件处理程序时使用 @EventListener。如果聚合具体地处理了一个事件，并且这个事件提供聚合的持久化标识符，那么在指定事件处理器的时候使用 @EventSourcingHandler。

当命令处理程序发布了一个要操纵聚合中特定实例的事件时，Axon 需要一种能够将该事件路由到聚合的正确实例的方法。事件可以提供标识符（注解 @TargetAggregateIdentifier），将其链接到聚合的特定实例。如果命令处理程序存在于一个集合中，那么 Axon 将从该存储库中加载该聚合的正确实例，并处理该聚合实例上的命令。





为什么要采用事件溯源？在我们的示例中，服务于客户服务的用例，保留投诉变化的历史很有价值。我们希望能够重播导致整体最终状态的事件。如果你更喜欢使用 Axon 存储库，而不是从数据库加载聚合，则将 `@Entity` 添加到聚合中，然后删除 `@AggregateIdentifier`，将其替换为 JPA 的 `@Id`。你也可以放弃 `@EventSourcingHandler` 注解的方法。

这就是 `ComplaintAggregate` 及其命令处理程序和事件处理程序。当 `FileComplaintCommand` 被发布时，用它创建该聚合的新实例。当 `AddCommentCommand` 被发布时会触发一个事件，该事件中包含 Axon 用来将处理路由回该聚合实例的目标 `@AggregateIdentifier`。Axon 将通过加载事件存储中的事件并调用相应的事件溯源处理程序来重新计算聚合，这些处理程序在重播时会产生反映该聚合的最新状态的聚合。

你可能已经注意到事件是命令的镜像，将命令式动词变成过去分词，例如，`FileComplaintCommand` 产生 `ComplaintFiledEvent`。这是典型的，但这不代表一个命令不能导致创建多个事件，从而启动更多的命令。当你在命令处理程序中调用 `AggregateLifecycle#apply` 方法时，此聚合上的事件溯源处理程序将自动被保存在事件存储区中。当 Axon 试图重建聚合时，它将知道以相同的顺序调用相同的事件溯源处理程序，以获得聚合的最新已知和有效状态。

我们来看事件。与命令一样，它们仅仅是包含很少内在状态的数据（如示例 12-8、示例 12-9、示例 12-10 所示）。

示例12-8 投诉提交时，`ComplaintFiledEvent`将被发送

```
package complaints;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class ComplaintFiledEvent {

    private final String id;

    private final String company;

    private final String complaint;
}
```

示例12-9 当评论被添加到父投诉时，`CommentAddedEvent`将会被分派

```
package complaints;

import lombok.AllArgsConstructor;
```




```
import lombok.Data;

@Data
@AllArgsConstructor
public class ComplaintAddedEvent {
```

```
    private String commentId;

}
```

示例12-10 投诉结束后，ComplaintClosedEvent将被分派

```
package complaints;
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
```

```
@Data
@AllArgsConstructor
public class ComplaintClosedEvent {
```

```
    private String complaintId;

}
```

在一个或多个查询模型中事件将更新绑定到命令模型上来进行更新。在查询（或视图模型）中，组件监听事件并更新相应的查询模型。在这种情况下，查询模型也是由 Spring Data JPA 的存储库驱动的，尽管它可以使用 MongoDB、HBase、Neo4J、文件系统上的文件或其他任何东西来处理。视图模型由两个 JPA 实体组成，即 ComplaintQueryObject 和 CommentQueryObject。ComplaintQueryObject 拥有一个 CommentQueryObject 实例的集合。ComplaintEventProcessor 将事件转换为查询模型上的操作，根据需要更新或保留 JPA 实体（如示例 12-11 所示）。

示例12-11 ComplaintEventProcessor

```
package complaints.query;
```

```
import complaints.CommentAddedEvent;
import complaints.ComplaintClosedEvent;
import complaints.ComplaintFiledEvent;
import org.axonframework.eventhandling.EventHandler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
import java.util.Collections;
```



```

@Component
public class ComplaintEventProcessor {

    private final ComplaintQueryObjectRepository complaints;

    private final CommentQueryObjectRepository comments;

    ❶
    @Autowired
    ComplaintEventProcessor(ComplaintQueryObjectRepository complaints,
        CommentQueryObjectRepository comments) {
        this.complaints = complaints;
        this.comments = comments;
    }

    @EventHandler
    public void on(ComplaintFiledEvent cfe) {
        ComplaintQueryObject complaint = new ComplaintQueryObject(cfe.getId(),
            cfe.getComplaint(), cfe.getCompany(), Collections.emptySet(), false);
        this.complaints.save(complaint);
    }

    @EventHandler
    public void on(CommentAddedEvent cae) {
        ComplaintQueryObject complaint = this.complaints
            .findOne(cae.getComplaintId());
        CommentQueryObject comment = new CommentQueryObject(complaint,
            cae.getCommentId(), cae.getComment(), cae.getUser(), cae.getWhen());
        this.comments.save(comment);
    }

    @EventHandler
    public void on(ComplaintClosedEvent cce) {
        ComplaintQueryObject complaintQueryObject = this.complaints.findOne(cce
            .getComplaintId());
        complaintQueryObject.setClosed(true);
        this.complaints.save(complaintQueryObject);
    }
}

```

❶ 这个事件处理程序使用了两个基于 Spring Data JPA 的存储库实现。

我们已经注意到，以上基本上是异步更新，并且独立于命令的处理。事件处理程序异步更新投诉域模型，并独立于分派事件的命令。我们也可以使用消息系统将这些事件发送给系统中的其他节点。

Axon 具有通过 AMQP (RabbitMQ 等代理支持的协议) 的事件分发功能。我们需要配置应用程序来与 RabbitMQ 实例进行通信。添加 Spring Boot 的 RabbitMQ starter `spring-boot-starter-amqp`, 并配置一个 RabbitMQ `ConnectionFactory` (你可以使用默认值或者在 `Environment` 中指定属性), Axon 将相应地广播事件。我们来看看投诉 API 发布事件所需的配置。首先, 我们需要在示例 12-12 中配置 `application.properties` 中的目的地的名字。

示例12-12 application.properties

```
axon.amqp.exchange=complaints
logging.level.org.axonframework=DEBUG
```

当然, 引用的 RabbitMQ 中的交换机 (`complaint`) 可能存在也可能不存在, 所以我们还需要配置 AMQP 组件。AMQP 使队列等代理端组件的创建和管理成为协议的一部分, 因此所有的 RabbitMQ 客户端都可以像示例 12-13 那样配置。

示例12-13 AmqpConfiguration

```
package complaints;
```

```
import org.springframework.amqp.core.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
```

```
class AmqpConfiguration {
```

```
    private static final String COMPLAINTS = "complaints";
```

❶

```
@Bean
```

```
Exchange exchange() {
```

```
    return ExchangeBuilder.fanoutExchange(COMPLAINTS).build();
```

```
}
```

❷

```
@Bean
```

```
Queue queue() {
```

```
    return QueueBuilder.durable(COMPLAINTS).build();
```

```
}
```

❸

```
@Bean
```

```
Binding binding() {
```

```
return BindingBuilder.bind(queue()) //
    .to(exchange()).with("").noargs();
}
```

④

```
@Autowired
public void configure(AmqpAdmin admin) {
    admin.declareExchange(exchange());
    admin.declareQueue(queue());
    admin.declareBinding(binding());
}
}
```

- ① 消息到达交换机。
- ② 然后将其转发给队列。
- ③ 使用绑定将两者链接在一起。
- ④ 并确保 RabbitMQ 相应地声明它们。

投诉统计 API

有了这个基础架构，将其他节点连接到这些事件上以便更新各自的查询模型就变得轻而易举了。demo-complaints-stats 模块是一个 REST API，它响应 ComplaintFiledEvent 事件来更新收到的被投诉的公司的记录。我们必须告诉 Axon 从 RabbitMQ 中排除事件，并将它们发送给相应的事件处理程序。Axon 公开了一个叫作事件处理器的概念，因为事件可能有不同的来源。我们指定的这个基于 AMQP 的处理器在配置中被称为 complaints（如示例 12-14 所示）。

示例12-14 application.properties

```
axon.eventhandling.processors.statistics.source=statistics
server.port=8081
```

最后，我们必须建立一个 AMQP 消息监听容器来处理传入的消息。我们必须使用示例 12-15 中的配置属性将传入消息绑定到基于 AMQP 的处理器上。

示例12-15 ComplaintsStatsApplication

```
package complaints;

import com.rabbitmq.client.Channel;
import org.axonframework.amqp.eventhandling.spring.SpringAMQPMessageSource;
import org.axonframework.serialization.Serializer;
```



```
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
```

```
@SpringBootApplication
```

```
public class ComplaintsStatsApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(ComplaintsStatsApplication.class, args);
    }
```

❶

```
@Bean
```

```
SpringAMQPMessageSource statistics(Serializer serializer) {
    return new SpringAMQPMessageSource(serializer) {
```

```
        @Override
```

```
        @RabbitListener(queues = "complaints")
```

```
        public void onMessage(Message message, Channel channel) throws Exception {
            super.onMessage(message, channel);
        }
```

```
    };
}
```

```
}
```

❶ 配置应该如何使用和分派入站事件。

这个简单的 REST API 中的主要监听器是在 REST API 本身中。它保存着每个公司观察到的投诉计数（并发友好）映射（如示例 12-16 所示）。

示例12-16 StatisticsRestController

```
package complaints;
```

```
import org.axonframework.config.ProcessingGroup;
import org.axonframework.eventhandling.EventHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.atomic.AtomicLong;
```

❶

```

@ProcessingGroup("statistics")
@RestController
class StatisticsRestController {
    //@formatter:off
    private final ConcurrentMap<String, AtomicLong> statistics =
        new ConcurrentHashMap<>();
    //@formatter:on

    ❷
    @EventHandler
    public void on(ComplaintFiledEvent event) {
        statistics.computeIfAbsent(event.getCompany(), k -> new AtomicLong())
            .incrementAndGet();
    }

    ❸
    @GetMapping
    public ConcurrentMap<String, AtomicLong> showStatistics() {
        return statistics;
    }
}

```

- ❶ 我们只需要来自 `statistics` 处理器组的事件消息。
- ❷ 当事件到达时，更新内存中的 Map。
- ❸ 然后将其暴露在 REST 端点下。向默认的 `http://localhost:8081/` 发出 HTTP GET。

目前，我们已经构建了两项服务，其中一项服务依赖于另一项服务，但是并没有耦合其他服务。混合引入更多的服务并不难。组件不需要知道或关心其他服务如何管理其状态。

到目前为止，我们只是开始使用 Axon 和 CQRS 来描述能做的事情。Axon 3 是一个彻底拥抱 Spring Boot 的巨大变革。Axon 已经整合了 Spring 生态系统的许多方面。希望在未来的版本中能够轻松地整合 Spring 的 `MessageChannel` 实例，而不是特别地耦合到 RabbitMQ。Axon 使 CQRS 以简洁一致的方式被轻松实现。你可以从同一个节点中的命令和查询模型开始，然后通过消息传递将工作量在多个节点上分布。你可以在没有事件溯源支持的情况下启动，如果出现用例，则将其集成。Axon 还提供了对 saga 的聚合的支持，这在本章的前面已经介绍过了，尽管这是另一个话题。

Spring Cloud Data Flow

在第 10 章中，我们研究了如何通过正确的抽象（Spring Integration 和 Spring Cloud Stream）使用 `MessageChannel` 来与其他系统协同工作。Spring Cloud Stream 根据约定和

外部配置将通过消息代理所需的连接剥离。对我们的服务来说，数据来自于通道也从通道中离开。通道是“管道与过滤器”架构中所说的管道，它们把不同的组件相互连接起来。这些组件的“接口”是统一的：Message <T>。这与命令行中的 stdin 和 stdout 工作方式非常相似：data-in 和 data-out。传输很好理解，是指所有的组件都同意互操作什么样的有效载荷来生产或消费。在 UNIX shell 环境中，通过单个集中的命令行实用程序，通过 stdin 和 stdout 管道数据，可以轻松地组合任意复杂的解决方案。我们可以使用基于 Spring Cloud 的消息传递微服务来做同样的事情。Spring Cloud Stream 提高了抽象层次，使我们可以专注于业务逻辑，忽略从一个服务到另一个服务的传输细节。我们可以创建更高阶的系统，并使用 Spring Cloud Data Flow 编排我们的消息传递微服务。

Spring Cloud Data Flow 支持轻松创建阶段式事件驱动架构（SEDA），这是云中理想的模式，服务需要足够强大，以吸收额外的负载和水平扩展。



什么是 SEDA？SEDA（Staged Event-Driven Architecture）是指将事件驱动的应用程序分解为一组由队列连接的阶段性的软件架构。它避免了与基于线程的并发模型相关的高额开销，并将应用程序逻辑中的事件和线程调度分离开来。可以通过限制每个事件队列的传入负载保持服务可用。这样可以防止在需求超过服务容量时资源被过度使用。有关更多信息，请参阅第 10 章中关于消息传递的讨论。

Spring Cloud Data Flow 的核心是流和任务的概念。

流（Stream）代表不同的基于 Spring Cloud Stream 模块应用程序的逻辑串联。Spring Cloud Data Flow 最终部署并启动不同的应用程序，并覆盖默认的 Spring Cloud Stream 绑定目标，以便数据按照我们描述的方式从一个节点流向另一个节点。复杂事件处理、传感器分析、日志分析、在线交易以及许多其他以事件为中心的场景使用 stream 来建模将变得很容易。

另一方面，任务（task）是我们想要检查是否达到我们期望状态的流程，如果不是则立即终止。

严格来说，Spring Cloud Data Flow 不会部署你的应用程序，它将这项工作交给 Spring Cloud Deployer 实现。在大多数数据分析平台（如 Apache Flink、Apache Spark 或 Apache Hadoop）定义它们自己的集群运行时环境的情况下，Spring Cloud Data Flow 团队竭尽全力确保其能够利用现有且功能非常强大的运行时环境。Spring Cloud Deployer 项目（<https://github.com/spring-cloud/spring-clouddeployer>）定义了一个抽象，用来管理任务或流在本地或运行时和平台的运行，如 Cloud Foundry（<http://cloudfoundry.org>）、Apache YARN、Mesos 或 Kubernetes。有 Hashicorp Nomad（<http://bit.ly/2s9VzpU>）

和 RedHat Openshift (<http://bit.ly/2salrSV>) 的社区贡献和维护实现。它提供了两个关键的接口：AppDeployer 和 TaskLauncher。AppDeployer 关注部署的生命周期，TaskLauncher 关心执行的生命周期。AppDeployer 支持部署、取消部署和查询（可能不确定的长期）应用程序的状态。TaskLauncher 支持启动、取消和清理如此部署的应用程序的实例。



在这个例子中，我们将看到 *local* Spring Cloud Data Flow 的实现，只需要将编译的 .jar 构件部署到本地 Maven 仓库即可。

要想查找你的分布式系统的 Data Flow 实现，可查看 Spring Cloud Data Flow 项目页面 (<http://cloud.spring.io/spring-cloud-dataflow/>)。你会发现许多技术的实现，包括 *Local Server*，我们将以它为例子。本章中代码的集成测试部署在 Spring Cloud Data Flow Cloud Foundry Server，并且使用它可实现真正的横向可伸缩性。

Stream

Stream 进一步被细分为三种类型的组件：源 (*source*，消息生成的地方)、处理器 (*processor*，传入的消息被处理后发送出去) 和接收器 (*sink*，传入的消息被消费的地方)。在 Spring Cloud Data Flow 中，这些组件在运行时都是独立的 Spring Cloud Stream 应用程序实例。源和接收器分别与 Spring Integration 中的入站适配器和出站适配器的概念相对应，而处理器差不多与 Spring Integration 中变换器的概念相对应。所有的流都由这三个组件组合而成。Spring Cloud Data Flow 自动将这些应用程序实例与 Spring Cloud Stream 绑定进行组合，并将一个正在运行的应用程序连接到另一个进程或另一个节点上。

默认情况下，Spring Cloud Data Flow 服务器是一个空白画布 (canvas)。它需要注册的应用程序（更复杂的解决方案的构建块）来组成自定义流和任务定义。在开始编写和注册自定义应用程序之前，请尝试使用 Spring Cloud Stream App Starters 项目 (<https://github.com/spring-cloud/spring-cloudstream-app-starters>) 提供的许多预构建应用程序的 starter。你可以一个一个地手动注册应用程序，或批量使用应用程序属性文件。Spring Cloud Data Flow 将在各种 Spring Cloud Deployer 实现中启动这些应用程序，并使用各种 Spring Cloud Stream 绑定器将它们连接起来。这意味着需要多个配置维度：应用程序描述是指 Maven 工件还是 Docker 存储库工件？应用程序描述应该指向使用 Spring Cloud Stream RabbitMQ 绑定器还是 Spring Cloud Stream Apache Kafka 绑定器编译的应用程序？令人高兴的是，在 Spring Cloud Stream App Starters 主页上 (<http://cloud.spring.io/spring-cloud-stream-app-starters/>) 有定义。

你可以对系统中的所有自定义应用程序进行编目，并将其注册到 Spring Cloud Data Flow 服务器。示例 12-17 所示是几个模块的简单的应用程序属性文件。每行指定组件的类型（源、接收器、处理器等），应用程序的逻辑名称，以及可以使用的类路径中 `org.springframework.core.io.Resource` 实现类的 URL。Spring Cloud Data Flow 提供了支持 Maven 存储库查找和 Docker 仓库查找等的新实现。

示例12-17 自定义应用程序定义的属性文件

```
source.account-web=maven://org.cnj:account-web:0.0.1-SNAPSHOT
sink.account-worker=maven://org.cnj:account-worker:0.0.1-SNAPSHOT
source.order-web=maven://org.cnj:order-web:0.0.1-SNAPSHOT
sink.order-worker=maven://org.cnj:order-worker:0.0.1-SNAPSHOT
source.payment-web=maven://org.cnj:payment-web:0.0.1-SNAPSHOT
sink.payment-worker=maven://org.cnj:payment-worker:0.0.1-SNAPSHOT
source.warehouse-web=maven://org.cnj:warehouse-web:0.0.1-SNAPSHOT
sink.warehouse-worker=maven://org.cnj:warehouse-worker:0.0.1-SNAPSHOT
```

如果没有一个符合你的要求，你自己来构建应用程序也很容易。

假设我们想将自定义应用程序插入流定义中。使用 Spring Initializr (<http://start.spring.io>)，选择你想使用的绑定（RabbitMQ 或 Apache Kafka），然后使用一个简单的处理器填充代码，该处理器接受输入的 String 消息，在消息正文前面增加一个 {，在后面增加一个 }，然后转发消息（如示例 12-18 所示）。

示例12-18 一个简单的Spring Cloud Data Flow处理器

```
package stream;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Processor;
import org.springframework.integration.annotation.MessageEndpoint;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;
```

```
@MessageEndpoint
```

①

```
@EnableBinding(Processor.class)
```

②

```
@SpringBootApplication
```

```
public class ProcessorStreamExample {
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(ProcessorStreamExample.class, args);
```

```

}

@ServiceActivator(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
public Message<String> process(Message<String> in) {
    return MessageBuilder.withPayload("{\"" + in.getPayload() + "\"}") ❸
        .copyHeadersIfAbsent(in.getHeaders()).build();
}
}
}

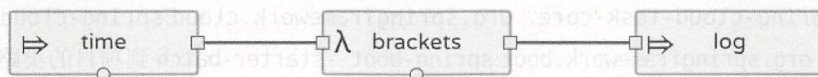
```

- ❶ 这是一个 Spring Integration 消息组件。
- ❷ 取决于默认的 Spring Cloud Stream 处理器绑定，它提供了一个众所周知的 input 和一个众所周知的 output MessageChannel 定义。
- ❸ 处理器用大括号包装其有效载荷。

我们已经将这个处理器应用程序安装到我们的本地 Maven 仓库中了，其 group ID 为 cnj，artifact ID 为 stream-example，版本为 1.0.0-SNAPSHOT。如果我们把这个处理器注册为一个应用程序定义文件中的应用程序，例如名为 brackets，那么我们可以在更复杂的解决方案中使用它。它将成为另一个构建块。将自定义应用程序和预建的应用程序混合，我们可以使用 Spring Cloud Data Flow 流 DSL 来组装更复杂的解决方案。假设我们希望将 time 源（每秒生成一个 Message）的输出连接到我们新导入的 brackets 处理器，然后将 brackets 处理器的输出连接到记录其输入的组件 log。Spring Cloud Data Flow 流 DSL 可以很容易地实现这样的流，如示例 12-19 和下图所示。

示例12-19 一个简单的流定义

time | brackets | log



这个定义将导致三个不同的操作系统进程被启动，它们可能在不同的节点上，这取决于正在使用的 Spring Cloud Deployer 实现。这些进程通过 Spring Cloud Stream binder（RabbitMQ、Apache Kafka 等）动态绑定。假设我们想要记录第一个流和 time 源的产出。我们可以在 Spring Cloud Data Flow 中使用分支来将一个组件（源或处理器）的产出路由到另一个组件，除了已经建立的任何连接外。我们来看看示例 12-20 中的结果如何。

示例12-20 分支时间源的结果

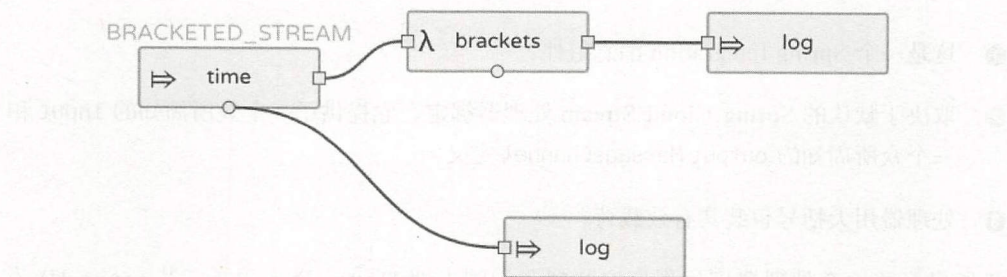
```

BRACKETED_STREAM=time | brackets | log ❶
:BRACKETED_STREAM.time > log ❷

```


- ❶ 将定义分配给变量名称。
- ❷ 然后解引用我们希望根据变量名称分支其输出的组件。

如下图所示。



流是无限量处理的理想选择。由于组件通过队列进行连接，因此可以使用 Spring Cloud Deployer 实现的分布式结构中可用的任何机制轻松扩展组件。在 Cloud Foundry 上，就像使用 `cf scale -i ..` 一样，其中 `..` 是在 Spring Cloud Data Flow 流定义中为组件分配的应用程序实例的逻辑名称。队列承担额外的负载，确保即使你没有足够的容量，整个系统也能保持稳定。Steam 是描述阶段式事件驱动架构（SEDA）的理想方法。

任务

Spring Cloud Data Flow 还可以管理基于 Spring Batch 的任务的生命周期和部署。使用 Spring Initializr 生成一个 Spring Cloud 应用程序。添加 `org.springframework.cloud:spring-cloud-task-core`、`org.springframework.cloud:spring-cloud-task-batch` 和 `org.springframework.boot:spring-boot-starter-batch` 到项目的类路径来激活 Spring Batch、Spring Cloud Task 和 Spring Cloud Task 桥接（如示例 12-21 所示）。

示例12-21 在Spring Cloud Task示例中定义的一个简单的Spring Batch Job

```
package task;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
```

```
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.task.configuration.EnableTask;
import org.springframework.context.annotation.Bean;
```

```
@EnableTask
```

```
①
```

```
@EnableBatchProcessing
```

```
@SpringBootApplication
```

```
public class BatchTaskExample {
```

```
    private Log log = LoggerFactory.getLog(getClass());
```

```
    public static void main(String[] args) {
```

```
        SpringApplication.run(BatchTaskExample.class, args);
```

```
    }
```

```
@Bean
```

```
Step tasklet(StepBuilderFactory sbf, BatchTaskProperties btp) {
```

```
    return sbf.get("tasklet").tasklet((contribution, chunkContext) -> {
```

```
        log.info("input = " + btp.getInput());
```

```
        return RepeatStatus.FINISHED;
```

```
    }).build();
```

```
}
```

```
@Bean
```

```
Job hello(JobBuilderFactory jbf) {
```

```
    Step step = this.tasklet(null, null);
```

```
    return jbf.get("batch-task").start(step).build();
```

```
}
```

```
}
```

① 激活 Spring Cloud Task 抽象。

在本地 Maven 仓库中安装这个任务应用程序，其中 group ID 为 cnj，artifact ID 为 task-example，版本为 1.0.0-SNAPSHOT，以备后用。

REST API

Spring Cloud Data Flow 的核心是提供 REST API 来管理流和任务的服务器。服务器将其状态保存在关系数据库中。它默认使用嵌入式的 H2 实例，不过你可以提供你自己的数据源定义。你可以通过 Spring Cloud Data Flow DataFlowTemplate、与 Data Flow 服务器捆绑在一起的 Web 应用程序或交互式 shell 来驱动此 API。

虽然可以将 Spring Cloud Data Flow 作为 Spring Boot 模块来启动，但建议你为在项目页面上 (<http://cloud.spring.io/spring-cloud-dataflow/>) 描述的每个 Spring Cloud Deployer 类型启用一揽子预构建的 .jar。下载当前版本，然后使用 java 命令运行 .jar。



运行 Cloud Foundry 版本时，除了还要指定一些帮助在 deployer 中使用 Cloud Foundry 客户端进行身份验证并与 Cloud Foundry 目标进行交互的属性（或环境变量）外，其他都是基本流程。

启动服务器，它将从端口 9393 启动，准备使用。

实现 Data Flow 客户端

随着服务器的启动，我们通常会做一些事情：

- 注册现有的任务和流应用程序（在 Maven 仓库或 Docker 仓库中）。
- 定义流和任务。
- 启动流和任务。
- 查询正在运行的流和任务的状态。

所以我们需要一个 Spring Cloud Data Flow 客户端！使用什么客户端完全取决于你要实现的目标。如果你是运维或开发人员，则可能需要从仪表板开始。它可以为你提供指导，它将 Spring Cloud Data Flow 所有有用的功能都放到了用户界面上。如果你想知道可以做什么，并希望有一个地方来试验，并获得快速的可视化反馈，那么就从仪表板开始。运维人员会更喜欢 shell，它提供了描述更高级别操作的脚本化方法。如果作为一名开发人员，你需要自动化 Spring Cloud Data Flow 的某些方面，或者将某些行为集成到一个更大的应用程序中，那么你可能会喜欢使用 DataFlowTemplate。

仪表板

从 <http://localhost:9393/dashboard> 访问仪表板 Web 应用程序。通过仪表板可以很容易地、直观地描述复杂的 Spring Cloud Data Flow 应用程序定义并对其进行管理。仪表板包含一个名为 Spring Flo 的可视化建模工具，它支持交互式构建流定义。

如果你想注册应用程序，请打开 APPS 选项卡（如图 12-1 所示）。

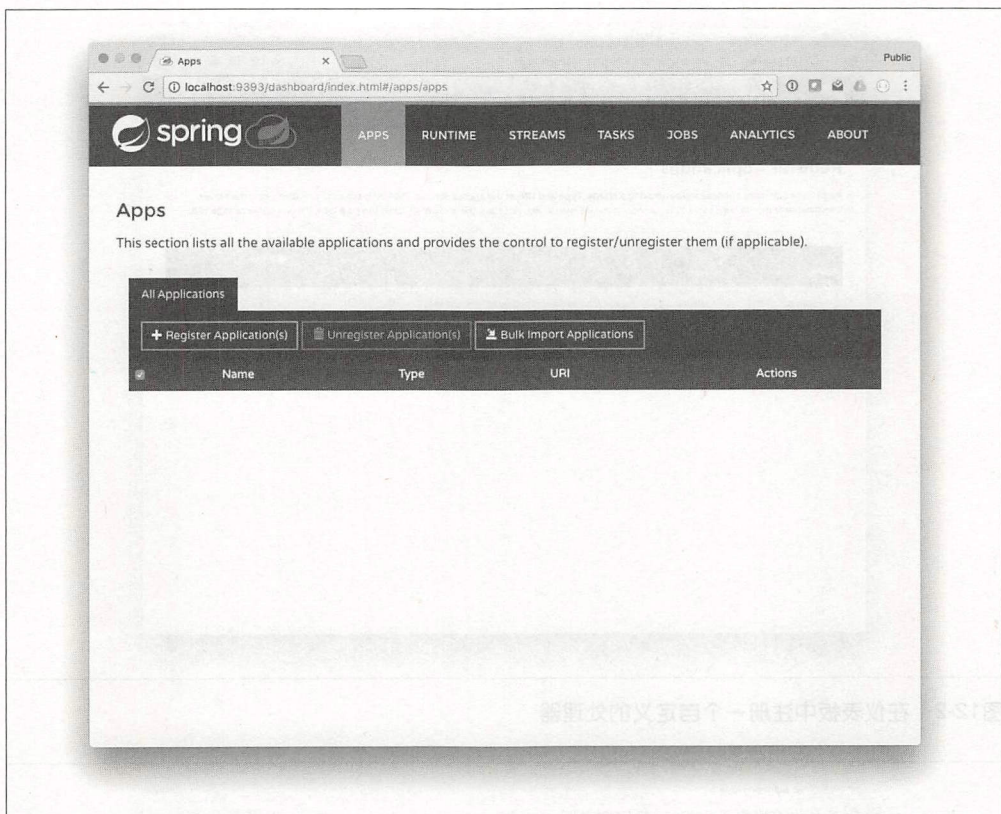


图12-1 Spring Cloud Data Flow仪表板的主页上的APPS选项卡

你可以将 Spring Cloud Data Flow 指向任何有效的 `org.springframework.core.io.Resource` 可解析的 URI 来定义应用程序。我们之前在 Maven 仓库中安装了一个 `processor` 和一个 `task` 应用程序。要安装 `processor`，为其指定一个名称，选择 `Processor` 作为应用程序类型，然后输入 URI：`maven://cnj:stream-example:1.0.0-SNAPSHOT`，如图 12-2 所示。

你可以通过引用应用程序定义文件或直接将内容粘贴到输入框中来批量注册多个应用程序，如图 12-3 所示。

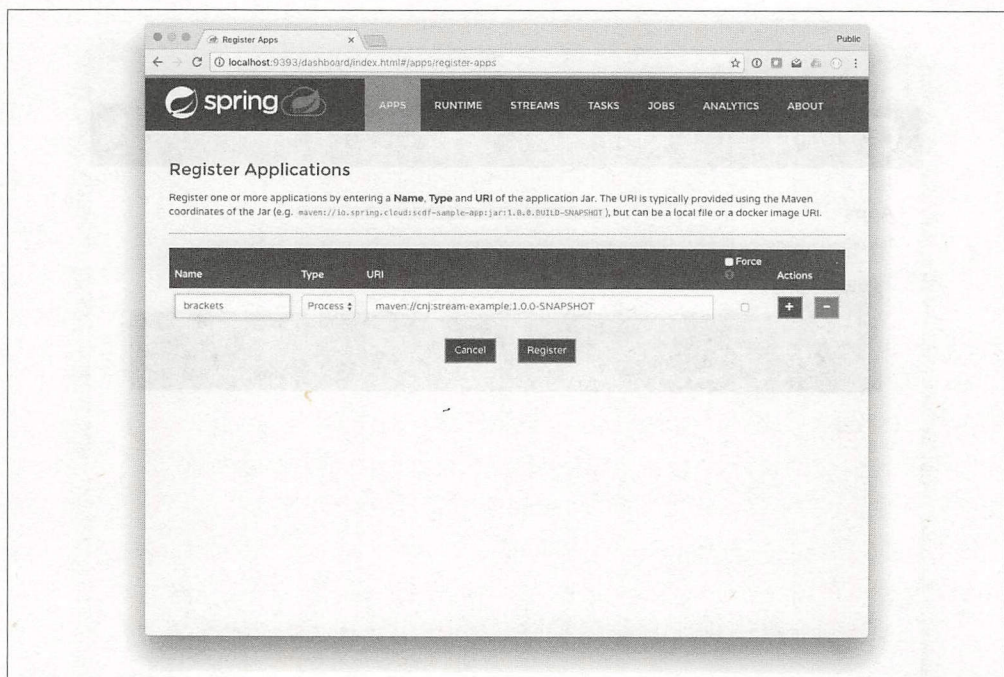


图12-2 在仪表板中注册一个自定义的处理器

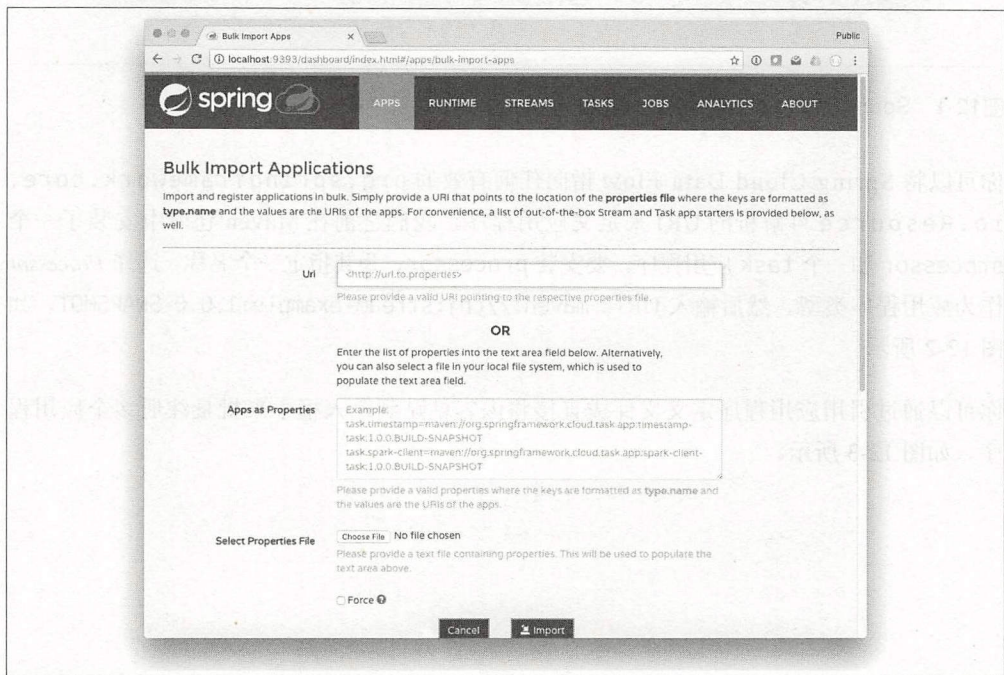


图12-3 应用程序定义的批量注册

你还可以选择批量注册 Spring Cloud Data Flow 团队提供的一些便捷的预先打包和预先编写的应用程序，并选择绑定和打包，如图 12-4 所示。

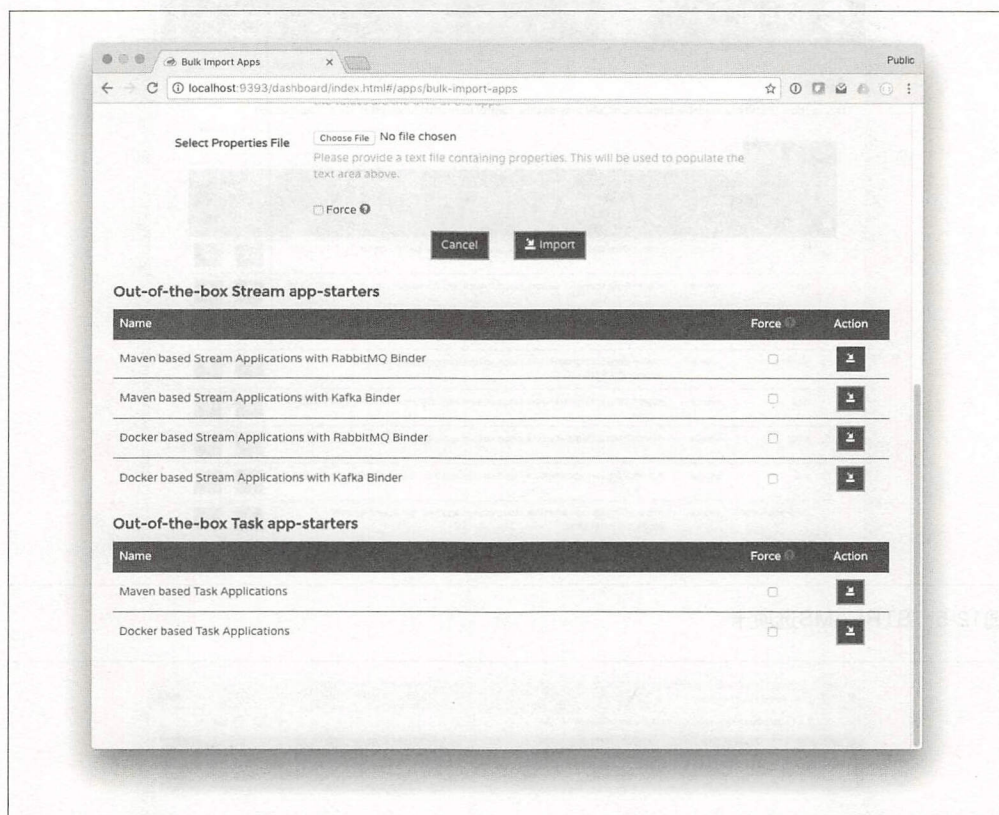


图12-4 使用开箱即用的Stream应用程序starter

现在你可以使用已注册的应用程序定义个性化流，如图 12-5 所示。

设计新的 stream 最简单（也是我们的首选）的方法是使用 Spring Flo 可视化设计器。设计师在可视化模型（你可以使用鼠标操作）和表单中的文本之间往返。可视化建模工具使用起来非常方便：从左侧的面板（source、sink 或 processor）中拖动一个项目，并使用其端口连接组件（小方块图标的左边或右边），如图 12-6 所示。

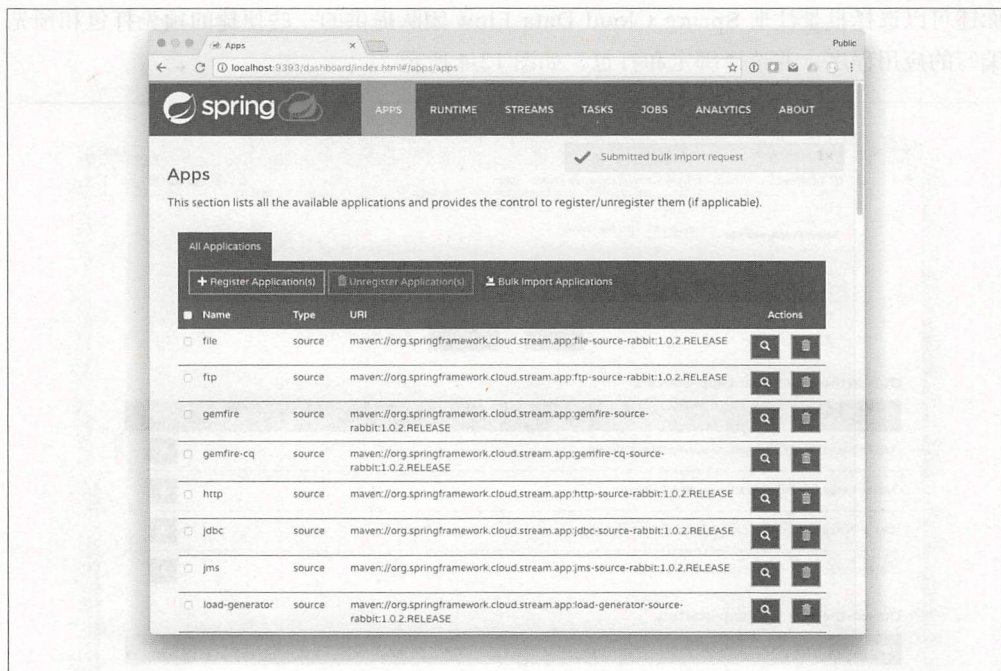


图12-5 STREAMS选项卡

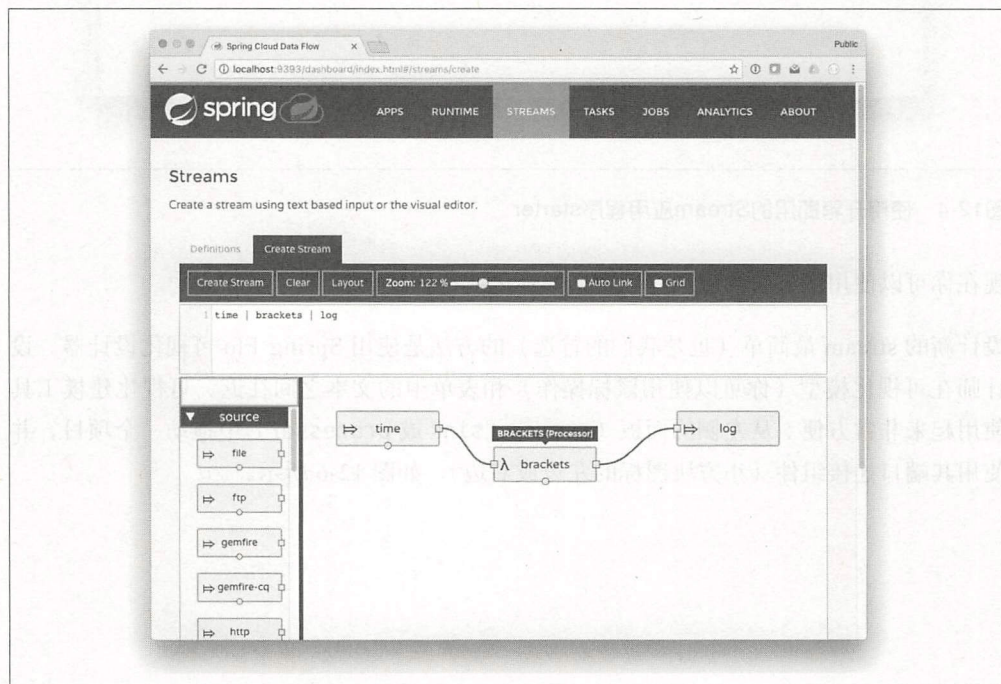


图12-6 设计stream

这样部署流定义已经很简单了。你需要给它起一个合乎逻辑的名字，如图 12-7 所示。

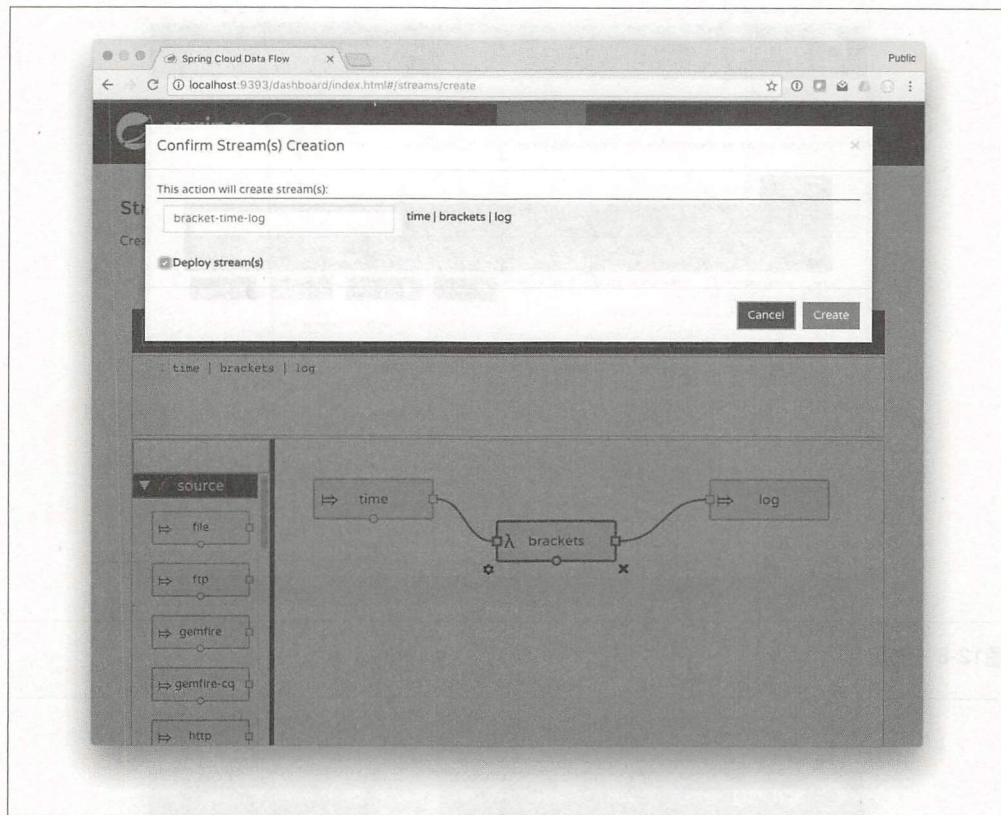


图12-7 给它取个名字

你将能够在 STREAMS 选项卡中看到已部署的流。单击箭头展开可视化项目，流是否已成功部署一目了然。如果你使用的是本地 deployer，请检查 Spring Cloud Data Flow 服务器的日志，以获取有关部署应用程序日志的位置信息，如图 12-8 所示。

除了不需要部署任务之外，这个基本工作流程同样适用于注册任务。你可以定义一个任务（基于已注册的任务应用程序）并启动一个或多个实例。单击相关任务旁边的 Create Definition 图标，如图 12-9 所示。

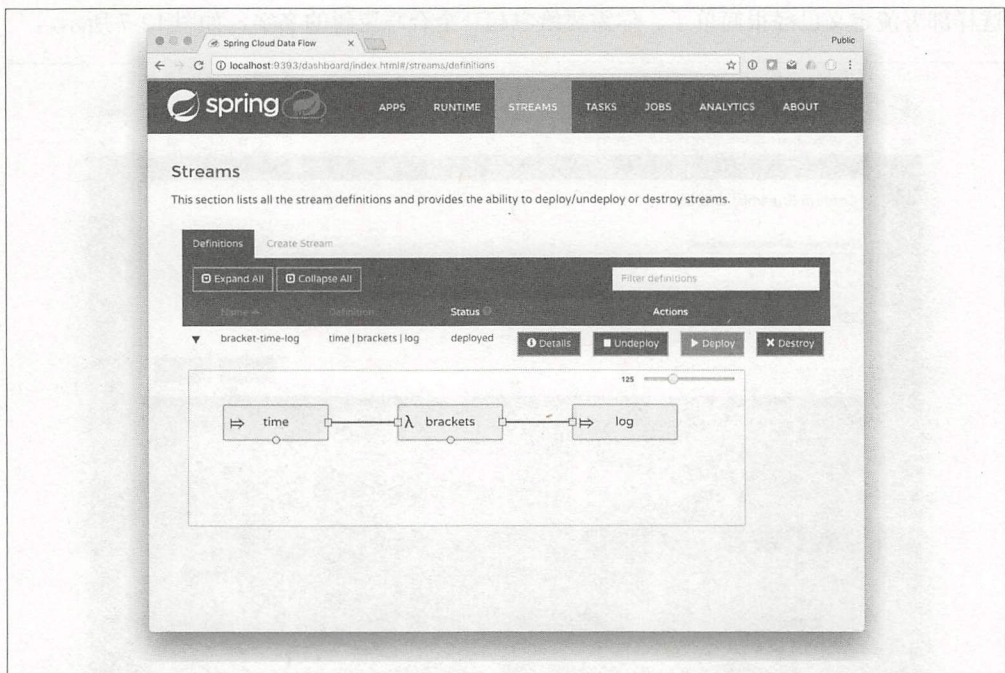


图12-8 流定义

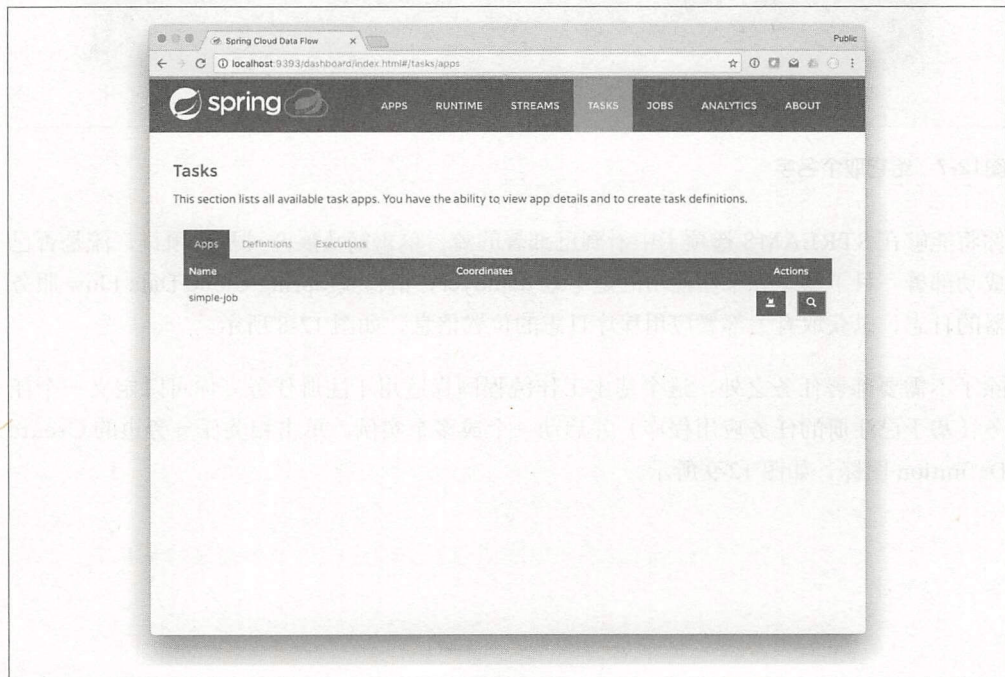


图12-9 简单工作任务

然后定义从一个运行到另一个运行的配置。你可能希望你的任务始终具有特定的 Actuator 设置，或特定的 Spring 配置文件。在这里指定它们，如图 12-10 所示。

The screenshot shows a web browser window titled "App Create Definition". The address bar shows "localhost:9393/dashboard/index.html#/tasks/apps/simple-job/create-definition". The form contains the following fields and options:

- idle-timeout**: Input field with value "30000". Description: "Connection idle timeout in milliseconds." ☐ Include
- Max-connections**: Input field with value "1". Description: "Maximum number of pooled connections." ☐ Include
- Enabled**: Input field (checkbox). Description: "Whether a PooledConnectionFactory should be created instead of a regular ConnectionFactory." ☐ Include
- Expiry-timeout**: Input field. Description: "Connection expiration timeout in milliseconds." ☐ Include
- Enabled**: Input field (checkbox). ☐ Include
- Mime-types**: Input field. ☐ Include
- Excluded-user-agents**: Input field. ☐ Include
- Min-response-size**: Input field. ☐ Include

Resulting Definition

simple-job --spring.profiles.active=scdf

Buttons: Back, Submit

图12-10 指定一个特定的Spring配置文件

你可以随时启动任务定义的实例，如图 12-11 所示。

当你启动一个任务的实例时，会被要求指定执行参数（作为参数传递给 `CommandLineRunner` 和 `ApplicationRunner`，并且作为作业参数传递给 `Spring Batch Job`），如图 12-12 所示。

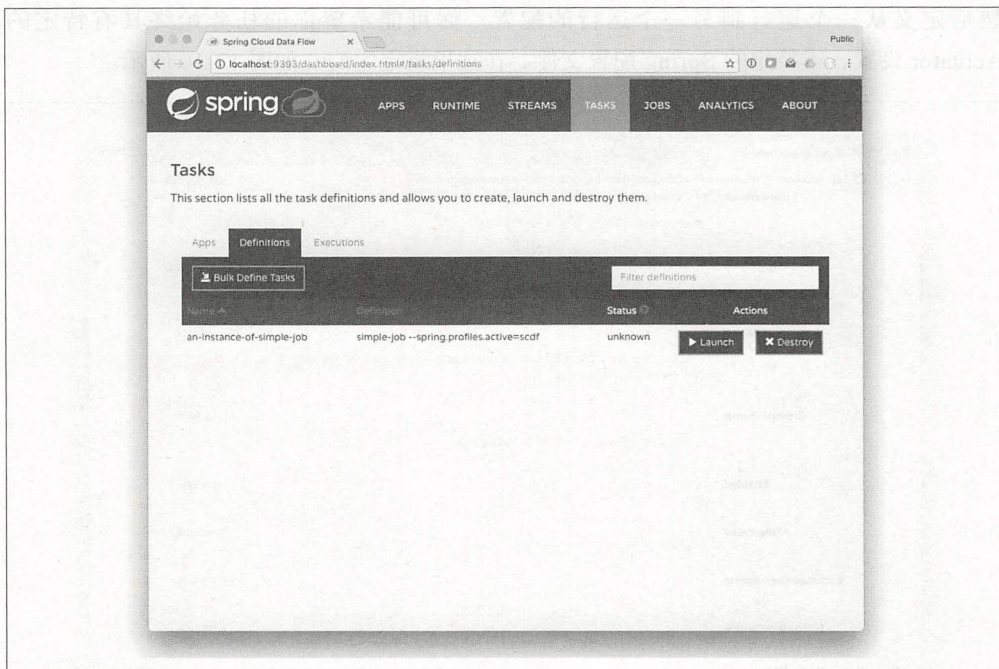


图12-11 查看任务定义

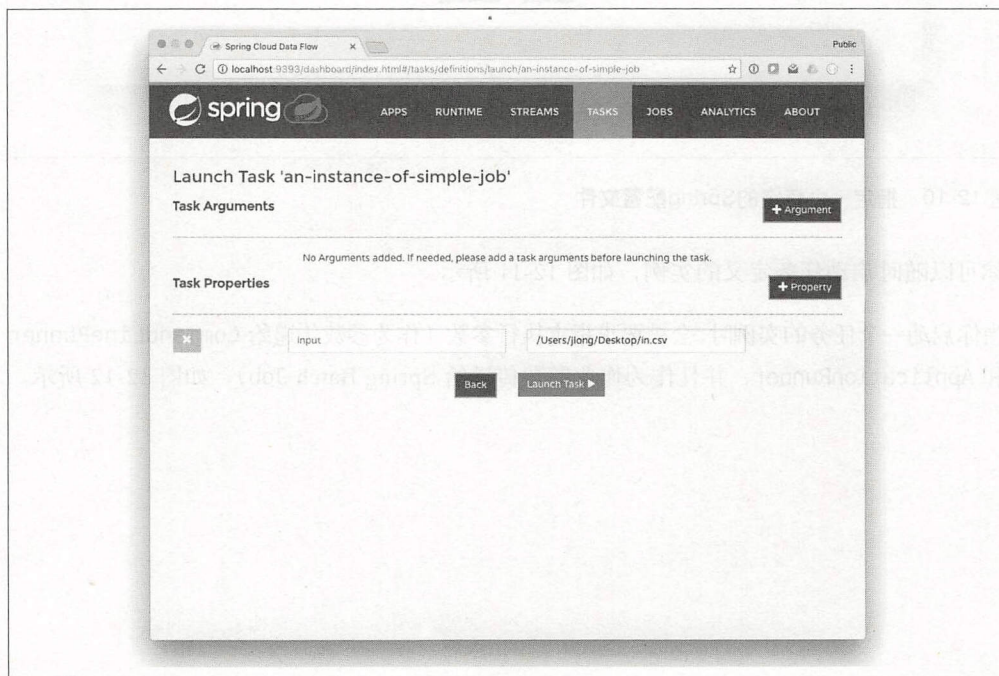


图12-12 指定实例参数和属性

最后，启动任务！图 12-13 是用户界面截图。

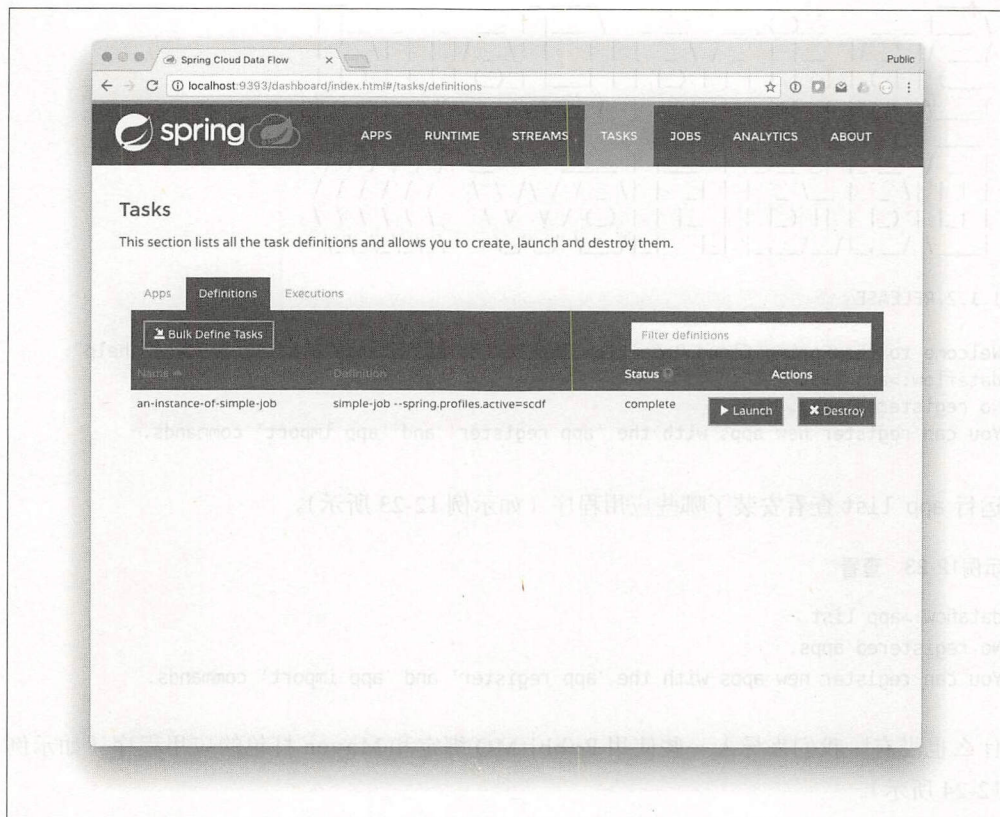


图12-13 查看任务及其状态

Spring Cloud Data Flow shell

如果你想使用一些更高级的功能，那就使用 Spring Shell (<http://bit.ly/2vHRPQM>) 的交互式 shell。参考 Spring Cloud Data Flow 页面 (<http://bit.ly/2vI40Nx>) 上的文档下载 shell 可执行文件。将 shell 下载到本地机器上后用 `java -jar spring-cloud-dataflow-server-local-....RELEASE.jar` 启动它，…是你下载的 shell 的相关版本。

运行 shell（如示例 12-22 所示）。

示例12-22 一个空的shell



1.1.2.RELEASE

```
Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type "help".
dataflow:>app list
No registered apps.
You can register new apps with the 'app register' and 'app import' commands.
```

运行 `app list` 查看安装了哪些应用程序（如示例 12-23 所示）。

示例12-23 查看

```
dataflow:>app list
No registered apps.
You can register new apps with the 'app register' and 'app import' commands.
```

什么也没有！我们来导入一些使用 RabbitMQ 绑定和 Maven 打包的应用程序（如示例 12-24 所示）。

示例12-24 导入RabbitMQ绑定的应用程序

```
dataflow:>app import --uri http://bit.ly/stream-applications-rabbit-maven
```

不错！我们使用两个默认应用程序创建一个简单的 `stream`：在计时器上产生一个新消息的 `time`；在 `stdout` 中记录传入消息的 `log`（如示例 12-25 所示）。

示例12-25 建立一个Spring Cloud Data Flow管道

```
dataflow:>stream create --name ticktock --definition "time | log" ❶
Created new stream 'ticktock'
```

```
dataflow:>stream deploy --name ticktock ❷
Deployed stream 'ticktock'
```

❶ 定义了一个需要产生时间的流，然后将它们传递给日志。

❷ 该流需要部署。调用底层的 `deployer`，然后启动该模块。在本地情况下，它基本上

在解析的模块的 .jar 上执行 `java -jar ..`。在 Cloud Foundry 上，它将为每个应用程序启动新的应用程序实例。

`time` 和 `timer` 都是完整的 Spring Cloud Stream 应用程序。它们公开了众所周知的通道：`time` 源的 `output`、`log` 接收器的 `input`，并且 Spring Cloud Data Flow 将它们拼接在一起，安排 `log` 应用程序的 `output` 通道被路由到 `time` 应用程序的 `input` 通道。在 Data Flow 服务器的控制台中，观察输出，确认相关的应用程序已启动并正在运行（如示例 12-26 所示）。

示例12-26 日志确认应用程序已经启动，并显示我们的各种日志

```
...
.. : FrameworkServlet 'dispatcherServlet': initialization completed in 11 ms
.. : deploying app ticktock.log instance 0
    Logs will be in /var/folders/cr/grkckb753fld3lbmt386jp740000gn/T/spring-.log
.. o.s.c.d.spi.local.LocalAppDeployer : deploying app ticktock.time instance 0
    Logs will be in /var/folders/cr/grkckb753fld3lbmt386jp740000gn/T/spring-.time
...
```

`log` 应用程序的 `stdout` 日志应该反映出一个永不停止的新时间戳流。

我们把定制的处理程序混合进去。我们需要先注册它（如示例 12-27 所示）。

示例12-27 使用shell注册和部署一个流

```
dataflow:>app register --name brackets --type processor \ ❶
--uri maven://cnj:stream-example:jar:1.0.0-SNAPSHOT
Successfully registered application 'processor:brackets'

dataflow:>stream create --name bracketedticktock --definition "time | brackets | log"
Created new stream 'bracketedticktock' ❷
```

```
dataflow:>stream list ❸
```

Stream Name	Stream Definition	Status
bracketedticktock	time brackets log	undeployed

```
dataflow:>stream deploy --name bracketedticktock ❹
Deployed stream 'bracketedticktock'
```

❶ 我们必须先注册应用程序，以便 Spring Cloud Data Flow 知道它的存在。

❷ 创建一个将 `time`、`log` 和新注册的 `bracket` 组件拼接在一起的流。

③ 确认流已被注册。

④ 部署流。

现在,我们启动一个简单的任务,看看是什么样,使用 timestamp 任务(如示例 12-28 所示)。

示例12-28 启动timestamp任务

```
dataflow:>task create --name whattimeisit --definition timestamp ①  
Created new task 'whattimeisit'
```

```
dataflow:>task list ②
```

Task Name	Task Definition	Task Status
whattimeisit	timestamp	unknown

```
dataflow:>task launch --name whattimeisit ③  
Launched task 'whattimeisit'
```

```
dataflow:>task list ④
```

Task Name	Task Definition	Task Status
whattimeisit	timestamp	complete

① 创建一个启动简单的 Spring Cloud Task 的任务,名为 timestamp 并打印出值。

② 列举现有的任务。

③ 启动任务,并给它取一个合乎逻辑的名称,稍后再引用它。

④ 列出任务。在这种情况下,任务是完整的。

检查日志,你应该能看到打印的时间。下面我们启动一个基于 Spring Batch 的任务(如示例 12-29 所示)。

示例12-29 使用shell注册并启动我们自定义的基于Spring Batch的任务

```
dataflow:>app register --name args --type task \ ❶  
--uri maven://cnj:task-example:jar:1.0.0-SNAPSHOT  
Successfully registered application 'task:args'
```

```
dataflow:>task create --definition "args --p1=1 --p2=2" --name args ❷  
Created new task 'args'
```

```
dataflow:>task launch --name args ❸  
Launched task 'args'
```

```
dataflow:>task list ❹
```

Task Name	Task Definition	Task Status
args	args	complete

❶ 我们必须先注册应用程序，以便 Spring Cloud Data Flow 知道它的存在。

❷ 创建应用程序的参数化的实例。

❸ 启动任务的一个实例。

❹ 注意任务的退出状态。

Spring Cloud Data Flow shell 功能很强大，因为它支持具有服务器和脚本功能的交互式会话。如果你需要更全面地控制，那么需要看看 `DataFlowTemplate`。

DataFlowTemplate

我们可以使用 `RestTemplate` 来处理 Spring Cloud Data Flow 的 API，以管理应用程序、流、任务、用户等。当访问 `http://localhost:9393` 时，这些 API 可以方便地作为超媒体链接被布局。我们还可以访问默认情况下在 `http://localhost:9393/management/` 下公开的 Spring Boot Actuator 端点。在完成身份验证、处理超媒体关系遍历等之后，我们可以通过一些努力达到目标。如果你使用的是 JVM，那么你可以使用 `DataFlowTemplate`，这是一个方便的客户端接口，通过它可以使用 Spring Cloud Data Flow 服务器中所有有趣的功能。将库添加到类路径下：`org.springframework.cloud:spring-cloud-dataflow-rest-client`。我们需要在 Spring Cloud Data Flow 服务器实现中初始化的所有东西，使用 `DataFlowTemplate` 在应用程序启动时安装将很方便。此初始化可以包括应用程序定

义、使用它们的流和任务以及这些流和任务的部署。下面是一个简单的组件，用于监听 `ApplicationReadyEvent` 应用程序的上下文事件（Spring Boot 在应用程序全部运行后发布的），并与新安装的服务器进行通信（如示例 12-30 所示）。

示例12-30 Data Flow初始化程序

```
package dataflow;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.cloud.dataflow.rest.client.DataFlowTemplate;
import org.springframework.cloud.dataflow.rest.client.TaskOperations;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import java.net.URI;
import java.net.URL;
import java.util.*;
import java.util.stream.Stream;

@Component
class DataFlowInitializer {

    private final URI baseUri;

    @Autowired
    public DataFlowInitializer(@Value("${server.port}") int port) {
        this.baseUri = URI.create("http://localhost:" + port);
    }

    @EventListener(ApplicationReadyEvent.class)
    public void deployOnStart(ApplicationReadyEvent e) throws Exception {

        ❶
        DataFlowTemplate df = new DataFlowTemplate(baseUri, new RestTemplate());

        ❷
        Stream
            .of("http://bit.ly/stream-applications-rabbit-maven",
                new URL(baseUri.toURL(), "app.properties").toString()).parallel()
            .forEach(u -> df.appRegistryOperations().importFromResource(u, true));
    }
}
```

```

3 TaskOperations taskOperations = df.taskOperations();
Stream.of("batch-task", "simple-task").forEach(tn -> {
    String name = "my-" + tn;
    taskOperations.create(name, tn);
    Map<String, String> properties = Collections.singletonMap(
        "simple-batch-task.input", System.getenv("HOME") + "Desktop/in.csv");
    List<String> arguments = Arrays.asList("input=in", "output=out");
    taskOperations.launch(name, properties, arguments);
});

```

```

4 Map<String, String> streams = new HashMap<>();
streams.put("bracket-time", "time | brackets | log");
streams
    .entrySet()
    .parallelStream()
    .forEach(
        stream -> df.streamOperations().createStream(stream.getKey(),
            stream.getValue(), true));
}
}

```

- ❶ 因为 `DataFlowTemplate` 在其构造函数中对 Data Flow 服务器进行 REST 调用，所以在 Data Flow 服务器初始化之后（在 `ApplicationReadyEvent` 的 `Spring ApplicationContext` 事件发布时）定义 `DataFlowTemplate`。确保 REST API 在调用之前可用。
- ❷ 引用应用程序定义的标准集合以及自定义任务和流定义，预注册应用程序定义。
- ❸ 注册并启动两个任务 `batch-task` 和 `simple-task` 的实例。
- ❹ 定义（并部署）一个名为 `bracket-time` 的流定义。

Spring Cloud Data Flow 让开发人员可以从较小、单一的组件构建高阶的系统。它针对应用程序和数据的可伸缩处理和集成进行了优化。从根本上说，Spring Cloud Data Flow 是云原生的批处理和消息传递。

总结

我们只是接触了一些数据处理的皮毛。这些技术中的每种技术都会涉及大量不同的技术。例如，我们可能会将 Apache Spark 或 Hadoop 与 Spring Cloud Data Flow 结合使用。我

们可以在 Spring Cloud Task 中使用 Activiti 或 Spring Statemachine。也可以将它们组合起来使用。例如，使用 Spring Cloud Stream 作为消息传递结构来扩展跨集群的 Spring Batch job 的处理。我们研究了支持结果恢复的技术，包括重试和恢复方法；以消息为中心的模式，如 saga 模式、命令查询责任分离（CQRS）和分阶段事件驱动架构（SEDA）。

第 IV 部分

生产

可观测的系统

建立一个有生产价值的系统有两个关键的技术方面：

- 我们必须建立一个足够可扩展的系统，并且能够处理业务需求。本书的大部分内容都侧重在这方面。我们研究了如何将分布式安全地引入到系统中的模式中，这反过来也使得应用对分布式系统的支持更加容易。我们已经在很大程度上摆脱了对容量规划的讨论，因为我们的主要目标是避免引入单点故障，而不是处理像 Google 那样的规模。一个被充分分解的系统在变成大规模时可能会遇到瓶颈，但届时要对具体问题进行分析。

当遇到这些需求时，要明确你需要的是可扩展性。表示系统可伸缩性的方式有很多种，除非确切知道系统的位置，否则无法达到你的要求。你可以使用这些测量结果来提供系统可扩展性的模型。有一个非常有用的模型被定义在通用可伸缩性定律 (<http://bit.ly/2sa2QpX>) 中，Baron Schwartz 在他的 *Practical Scalability Analysis with the Universal Scalability Law* (<http://bit.ly/2s9U23m>) (具有通用可伸缩性定律的实用可伸缩性分析) 一书中描述了该模型。Coda Hale 创建了一个有用的建模库 `usl4j` (<http://bit.ly/2s9U23m>)，给出了关于 Little's law (<https://github.com/codahale/usl4j>) 任何两个维度的观察结果——给你提供一个从任何其他维度预测任何维度的模型。这样的模型支持缩放以满足特定的需求。像 Cloud Foundry 这样的云平台可以让做这件事更轻松，因为它支持轻松的水平缩放。我们将在本章中介绍如何测量系统。这些度量将发送给可伸缩性模型。

- 我们必须建立一个系统，当事情不能按预期工作时系统可以做正确的事情。应用程序必须易于修复。

这两个要求都要求系统对我们是可见的——我们必须能够测量系统。除了技术要求之外，可视性也支持业务。如果我们以一种敏捷的方式进行真正的迭代，那么在每次迭代之后，对客户来说软件必须是可交付的，并且是可发布的，如果不发布，那么“客户”可能是客户端、非营利性组织、开源项目或你自己。

“客户”描述软件的价值。软件越早发布给客户，就会越早释放价值。发布的软件通过获得商业价值而持续发展。已发布且正在工作中的软件通过收获商业价值而持续发展。

我们如何知道软件是否正在工作？软件是沉默的。不论软件是生是死它都是安安静静的。没有指示其故障的信号或气味。当我们构建软件时，我们需要建立一个正在工作中的定义。这有助于我们设定正常操作行为的预期——一个基准（baseline）。基准是衡量行为改善的基础。

在本章中，我们将了解如果实现它们，所有应用程序需要具有哪些非功能或跨功能能力。本章的大部分内容涉及可观察性——如何通过系统的外部输出来判断其内部状态。

这些能力并没有业务差异。这不是你的组织商业化后才需要解决的！它们对应用程序的持续安全运行和演变至关重要。迈克尔·尼加德（Michael Nygard）在他的史诗般的著作 *Release it!*（Pragmatic）中详细描述了这些功能和你的关注点。问题的关键是代码完整性与生产就绪的含义不同。如果软件不是生产就绪的，则不能发布。

你构建，你运行

开发者从本能上一直忽略这些非功能性需求。毕竟，如果它们对功能集没有贡献，它们的存在也不会吸引客户，为什么要关注这些东西呢？当然，在项目开始的时候，它们不应该成为一个令人担忧的问题！开发人员在增加业务功能和处理不断积压的需求时，不希望因这些功能引起的代码更改而烦恼。如果软件在深夜里失败了，那也不是他们的问题；帮助台会处理！没有哪个帮助台或运营团队希望在凌晨 4 点黑盒子软件（完全没有可见性）出现故障，所以他们特别关心软件的可观察性，并尽其所能，在不改变代码的情况下支持它。其结果是开发人员会编写软件，并将其放在一个想象的墙上进行操作，这些操作会在不改变代码的情况下被度量。

如果开发人员和运维人员都意识到了彼此之间的关系，那么开发人员通常会因为应用程序的运行方式而感到自主权的丧失。开发人员想要发布代码，而运维人员则希望确保生产系统的稳定性。这样的关系并不理想。这导致业务成果与系统稳定性之间的脱节。今天，当然，我们讨论 *DevOps*。运维和开发人员都有责任确保系统和业务成果的稳定性。他们是同一枚硬币的正反面，有着共同的目标。许多高效能组织采用了一个简单的口头禅，以此来缩短开发者与运维者之间的隔阂：“是你构建的那就由你来运行。”现在团队负责维护生产中的代码。

如果你知道因为这些系统没有支持可观察性和修复的故障点，自己可能会在凌晨 4 点被唤醒来支持它们，那么你可能会对构建更具健壮性的系统感兴趣。

谋杀神秘微服务

在云中构建分布式系统时，支持可观察性的需求变得更为重要。有了微服务，每一次中断都更像是谋杀之谜 (<http://bit.ly/2sa2XBT>)。Increment Magazine (这是一个引人入胜的 on-call 运维新事物) 查看“当寻呼机关闭时会发生什么？” (<http://bit.ly/2s9XZVP>)。其描述了一个相当标准化、高层次的框架，整个行业的一大批高效能组织响应一个事件。下面是基本步骤。

分流

在系统的某个地方发生了错误。在这个阶段的工作是确定哪里可能出现故障，评估事件的影响和严重程度，并对事件进行分类。

协商

这个阶段的工作是准备开始缓解。这项工作可能由另一个团队（开发人员）或者对事件进行分类的同一个团队完成。通常，通过建立的渠道（聊天、Slack、Skype、Google 环聊等）进行沟通。正在进行的工作必须被记录和分享。许多组织创建了作战室并设置了电话会议。

缓解

在这个阶段，目标是要减轻事件的影响，恢复系统的稳定性，而不是找到根本原因，不是解决根本问题。例如，如果系统因部署而失败，则缓解步骤将是回滚更改，而不是尝试修复潜在的问题。

解决

缓解可能只能阻止事件的发展，阻止事件进一步影响其他用户，但不能解决根本问题。在解决阶段，开发人员进行根本原因分析并解决问题。现有用户可能仍然受到影响，该问题需要解决。时间在这里是至关重要的，但团队必须小心遵守正常的质量保证措施，如测试任何修补程序。许多组织会衡量平均缓解时间（MTTM）和平均解决时间（MTTR）。

跟进

在这个阶段，组织会试图通过免责的事后教训复盘，制定和分配后续任务，并举行事件复盘会议。这个事件只有在所有后续任务完成之后才被考虑。

在每一步中，组织的支持、人与人之间的沟通以及系统本身的可见性是非常重要的。

十二要素运维

可运维的应用程序就是为生产而构建的应用程序。正如我们将在本章中看到的，这样的

应用程序在业务功能之外将使用多样化的工具生态（集中式日志处理、健康和流程管理器、作业调度程序、分布式跟踪系统等）。

我们将通过，构建应用程序，从十二要素宣言的关键原则出发，使用 Spring Boot 和 Spring Cloud 的约定，从这个基础架构中受益。但是，请不要误解，需要有人提供该基础设施。没有配套的基础设施，盲目地部署到生产是不可行的。如果这个十二要素宣言描述了一套用户可用于在云上构建生产就绪的应用程序的良好、整洁的原则，那么需要有人满足合同的另一方，并支持 Andrew Clay Shafer (<https://twitter.com/littleidea>) 提出的十二要素运维。需要将应用程序和服务作为无状态进程运行，提供众所周知的应用程序生命周期，使外部化配置变得容易，支持日志管理，提供支持服务，使应用程序水平扩展，提供声明式端口绑定等。显然，Cloud Foundry 在支持运维要求方面做得非常好。

在本章中，我们将介绍如何逐个节点地显示信息，以及如何集中这些信息，以构建支持在一个统一平台上快速理解系统行为的体验。不仅仅是系统中的应用程序，捕获系统本身的行为也非常重要。

地图不是地形。仅仅查看曼哈顿的地图比起实际走过曼哈顿的街道简直就是走马观花，在系统的架构图中有很多无法捕获的可能导致严重问题的系统行为。这些行为只有通过有效的全系统监测才能获得。

新方式

将应用程序成功部署到生产环境的要求并没有太大改变。即使离职的开发者能够承担起运营问题，但组织要蓬勃发展，对应用程序中可能危及系统稳定性的操作无动于衷，那离职的开发人员如何能够负担得起。开发者和运维者之间的交接过去是不透明的交付物，就像一个兼容 Servlet 容器的 .war。在这个黑盒子中部署的应用程序受益于一些容器提供的服务，如内置的启动和停止脚本以及中央日志输出。运维人员需要进一步定制这个盒子，以使这些容器的承诺更有意义。然后，运维人员需要确保整个基础设施支持到位，以使应用程序在生产中保持稳定。

进程调度和管理

什么组件用来启动应用程序，并优雅地关闭它？如何确保它不在同一个主机上运行两次？

应用程序运行状况和修复

运维人员如何知道应用程序是否运行良好？如果应用程序进程死亡会发生什么？如果主机本身死亡会发生什么？

日志管理

运维人员如何查看从应用程序实例后台打印的日志？如何收集和分析？

应用程序可见性和透明度

运维人员如何捕获应用程序状态，或将应用程序状态量化为度量标准，并对其进行分析和可视化？

路由管理

应用程序是否暴露于互联网？负载均衡？在其他主机上重新启动应用程序时，路由是否正确更新？

分布式跟踪

谁在访问系统？处理交易时涉及哪些服务，这些请求的延迟是多少？平均调用图表是什么样的？

应用程序性能管理

运维人员如何诊断和解决复杂的应用程序性能问题？

“一个坏系统会打败一个好人！”

——W. Edwards Deming

Deming 说，如果换成在人类的背景下，而不是关于分布式系统，这些同样适用。如果用来管理软件的系统难以做正确的事情，那么人类也就难以做正确的事情。

尽可能无阻力地支持可观测性和这些最佳实践是非常重要的，因此在整个服务和项目中一致地引入可观测性是很不容易的。组织不论大小都知道可怕的公司维基（Wiki）页面（“迈向生产的 500 个简单步骤”），即在部署服务之前需要完成的模板和各种手动工作。Cloud Foundry 和 Spring Boot 给予了我们坚定的支持。Cloud Foundry 支持所有十二要素运维要求，而 Spring Boot（以及我们在讲解 Spring Boot 基础知识时讨论的自动配置）是对十二要素原则的具体实现和定制。它们将做正确的事情的认知开销降到几乎为零。我们只需要做一次，然后重用即可。不加选择地重复工作是速度的敌人。

这样的基础设施很难获得相应的开发成本投入。这也是非常重要的，但这不是业务功能。我们相信像 Heroku 这样的独立平台或开源 Cloud Foundry 的发行版提供了支持基础设施和易用性的最佳组合。这些平台对应用程序进行了一定的假设——它们是事务性的在线 Web 应用程序，符合十二要素宣言的原则。这些假设使平台很容易达到某些要求，它们减少了代码库中的变更。这些假设可以支持自动化并提高软件开发速度。

可观测性

可观测性是一个棘手的问题。如果你只有一个独立的应用程序，那么事情会更容易。当系统出现问题时，至少可以发现错误发生在哪里：错误来自系统内部！在分布式系统中事情明显变得复杂了，因为组件之间的交互使得故障隔离成为关键。在没有可以支持可视性的仪器、仪表和窗口的情况下你不会驾驶一辆客车，我们怎么可能希望在没有可视性的情况下对数百架飞机进行空中交通管制呢？

通过提升系统的可观测性，企业将可以对系统进行持续的投资以改善系统。运维人员通过良好的可观测性，至少可以发现系统中潜在的问题，并将这些问题与所发生的事件联系起来。在某些情况下，运维人员也可以使用良好的可观测性来自动响应或简化对事件的响应。

在理想情况下，系统的可运维性与业务的可见性可以相互关联，并用于产生一个“单一面板”的体验——仪表板。在本章中，我们将介绍如何收集和理解历史和现在的状态，以及如何支持前瞻预测，特别是基于历史数据的模型驱动的预测。

历史数据是一段时间内存储在某处的数据。由历史数据可以产生长期的业务洞察力，更新的数据可能支持运维、错误分析和调试。

系统是一个拥有大量活动部件的复杂机器。我们很难知道要收集哪些信息，忽略哪些信息，所以要谨慎，尽量收集各种信息。

推与拉的可观测性和解析率

有些监控和观测工具采用基于拉的方法，其中集中式的基础设施每隔一段时间从服务中拉取数据，还有些监控基础设施则预期所有节点会将其状态事件信息推送给它。在本章中，我们可以看到，许多工具使用其中一种方式或者两者兼而有之。这个你可以自己选择。

对于很多组织来说，解析率是一个需要讨论的问题。以什么样的频率更新监控基础设施？在动态环境中，所有事情都可能随着需求而变化。事实上，当我们谈论临时任务时，服务的寿命可能只有几秒或几分钟。如果系统使用基于拉的监控，则两次拉之间的时间间隔可能比正在运行的应用程序的整个生命周期的跨度还长！监控基础设施就像个盲人，看不到整个组件的运行情况，并可能错过数据中的高峰和低谷。这是接受对这些组件进行基于推式的监控的强有力的理由。

我们从 Spring 的灵活性中大大受益：它经常为我们提供可以用来触发监控事件的事件。在阅读本章时，问问自己，一个给定的方法是基于拉取还是基于推送的方式，并想想如

果需要的话，你应该如何设置拉取或推送。

使用 Spring Boot Actuator 捕获应用程序的当前状态

应用程序的当前状态就是你要映射到仪表板上的那些信息，也可以是办公室的大屏幕上人们可以看到的信息。你也可能会保留所有这些信息供以后使用。当前状态就像汽车里的速度表，它应该尽可能简明扼要地告诉你是否有麻烦。

如果必须将系统的状态提取为可视化信息（红色表示危险；绿色表示一切正常；或者黄色表示可能存在某些问题，但在可接受的范围内），那么你会选择哪些信息？这就是当前的状态信息。

当前状态可能包括内存、线程池、数据库连接和处理的总请求等信息。它可能包括每秒请求数（系统所有部分的请求，包括 HTTP 端点和消息队列）、响应时间的 95% 统计、遇到的错误以及断路器状态等统计信息。

Spring Boot Actuator 框架通过端点提供了对有关应用程序信息的开箱即用支持。端点收集信息并有时与其他子系统交互。这些端点可以通过许多不同的方法查看（例如使用 REST 或 JMX）。我们将重点讨论 REST 端点。端点是可插拔的，各种基于 Spring Boot 的子系统通常会在适当的时候提供额外的端点。要使用 Spring Boot Actuator，请将 `org.springframework.boot:spring-boot-starter-actuator` 添加到你的项目构建中。添加 `org.springframework.boot:spring-boot-starter-web` 以将自定义端点暴露为 REST 端点。



下面是摘自 Spring Boot 文档的一段话 (<https://docs.spring.io/springboot/docs/current/reference/htmlsingle/>): “actuator (执行器) 是一个制造业术语，指的是用于移动或控制某些东西的机械设备。执行器可以从一个小小的变化中产生大量的动作。”

表 13-1 列出了一些 Actuator 端点。

表13-1 Actuator端点

端点	用法
/info	公开有关当前服务的信息
/metrics	公开有关服务的量化值
/beans	公开 Spring Boot 为你创建的所有对象的图形

端点	用法
/configprops	公开有关可用于配置当前 Spring Boot 应用程序的所有属性的信息
/mappings	公开 Spring Boot 在这个应用程序中知道的所有 HTTP 端点以及任何其他元数据（例如 Spring MVC 映射中指定的内容类型或 HTTP 动词）
/health	系统中组件状态的描述：UP、DOWN 等。同时返回 HTTP 状态码
/loggers	显示和修改应用程序中的记录器
/auditevents	显示由 AuditEventRepository 记录的所有 AuditEvent 实例。它们将认证的 Principal 实体连接到系统中的事件。你也可以捕获并发送自定义事件
/cloudfoundryapplication	公开基于 Cloud Foundry 的管理 UI 中的信息，并通过 Spring Boot Actuator 信息进行扩充。除了“运行”或“停止”以外，应用程序状态页面可能包含完整的 Spring Boot /health 输出。这些信息是安全的，并且需要来自 Cloud Foundry 的 UAA 认证和授权服务的有效令牌。如果你的应用程序未在 Cloud Foundry 上运行，则可以使用 management.cloudfoundry.enabled=false 来禁用此端点
/env	返回所有已知的环境属性，例如操作系统的环境变量或 System.getProperties() 的结果

下面我们来更深入地探讨这些端点。

度量

每个人都从太少的数据点概括，至少我是。

——Parand Darugar (<https://twitter.com/parand/status/854793437043761152>)

度量(metrics)是数字。在 Spring Boot Actuator 框架中,有三种度量标准:公开度量标准(我们稍后会看到)、量表(gauge)和计数器(counter)。量表记录一个单一的值,不需要制表。计数器记录增量(增量或减量),这是一个随着时间的推移而达到的值。度量是数字,因此易于存储、图形化和查询。有一些操作度量操作本身就会关心:特定于主机的信息,如 RAM 使用、磁盘空间和每秒请求数。组织中的所有人都会关心语义指标:在最近一小时内完成了多少次订单,发出了多少次订单,发生了多少次新账户注册,销售了哪些产品以及多少次,等等。默认情况下, Spring Boot 在 /metrics 端点(见示例 13-1)公开这些度量。

示例13-1 来自应用程序/metrics端点的度量

```
{
  "classes" : 9731,
  "heap.committed" : 570368,
  "nonheap.used" : 72430,
  "systemload.average" : 3.328125,
  "gauge.response.customers.id" : 7,
  "gc.ps_marksweep.count" : 2,
  "nonheap" : 0,
  "counter.status.200.customers" : 1, ❶
  "counter.status.200.customers.id" : 2,
  "mem.free" : 390762,
  "heap.used" : 179605,
  "classes.unloaded" : 0,
  "gauge.response.star-star.favicon.ico" : 4,
  "instance.uptime" : 47231,
  "counter.status.200.star-star.favicon.ico" : 2,
  "threads.peak" : 21,
  "nonheap.init" : 2496,
  "threads.totalStarted" : 27,
  "mem" : 642797,
  "httpsessions.max" : -1,
  "counter.customers.read.found" : 2,
  "gc.ps_marksweep.time" : 96,
  "uptime" : 52379,
  "threads" : 21,
  "customers.count" : 6,
  "gc.ps_scavenge.count" : 6,
  "heap.init" : 262144,
  "httpsessions.active" : 0,
  "nonheap.committed" : 74112,
  "gc.ps_scavenge.time" : 87,
  "counter.status.200.admin.metrics" : 2,
  "datasource.primary.usage" : 0,
  "processors" : 8, ❷
  "gauge.response.customers" : 9,
  "heap" : 3728384,
  "gauge.response.admin.metrics" : 4,
  "threads.daemon" : 19,
  "datasource.primary.active" : 0,
  "classes.loaded" : 9731
}
```

- ❶ 这些指标包括请求数、路径和 HTTP 状态码。这里的 200 是 HTTP 状态码。
- ❷ Spring Boot Actuator 还可以捕获有关系统的其他重要信息，例如可用的处理器数量等。

度量已经为我们记录了很多有用的信息：记录了所有的请求（和相应的 HTTP 状态码）以及关于环境的信息（如 JVM 的线程、加载的类及关于所有配置的 DataSource 的实例信息）。Spring Boot 根据正在使用的子系统有条件地注册度量。

你可以使用 `org.springframework.boot.actuate.metrics.CounterService` 来记录增量（“又有一个请求已经被创建”）或者 `org.springframework.boot.actuate.metrics.GaugeService` 捕获绝对值（“有 140 个用户连接到聊天室”），如示例 13-2 所示。

示例13-2 使用CounterService收集客户指标

```
package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.CounterService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import java.net.URI;

@RestController
@RequestMapping("/customers")
public class CustomerRestController {

    private final CounterService counterService;

    private final CustomerRepository customerRepository;

    @Autowired
    CustomerRestController(CustomerRepository repository,
        CounterService counterService) { ❶
        this.customerRepository = repository;
        this.counterService = counterService;
    }

    @RequestMapping(method = RequestMethod.GET, value =("/{id}")
    ResponseEntity<?> get(@PathVariable Long id) {
        return this.customerRepository.findById(id).map(customer -> {
            String metricName = metricPrefix("customers.read.found");
            this.counterService.increment(metricName); ❷
            return ResponseEntity.ok(customer);
        }).orElseGet(() -> {
            String metricName = metricPrefix("customers.read.not-found");
            this.counterService.increment(metricName); ❸
            return ResponseEntity.class.cast(ResponseEntity.notFound().build());
        });
    }
}
```



```

}

@RequestMapping(method = RequestMethod.POST)
ResponseBody<?> add(@RequestBody Customer newCustomer) {
    this.customerRepository.save(newCustomer);
    ServletUriComponentsBuilder url = ServletUriComponentsBuilder
        .fromCurrentRequest();
    URI location = url.path("/{id}").buildAndExpand(newCustomer.getId()).toUri();
    return ResponseEntity.created(location).build();
}

@RequestMapping(method = RequestMethod.DELETE)
ResponseBody<?> delete(@PathVariable Long id) {
    this.customerRepository.delete(id);
    return ResponseEntity.notFound().build();
}

@RequestMapping(method = RequestMethod.GET)
ResponseBody<?> get() {
    return ResponseEntity.ok(this.customerRepository.findAll());
}

④
protected String metricPrefix(String k) {
    return k;
}
}

```

- ❶ CounterService 是自动配置的。如果你使用的是 Java 8，那么执行效果会比早期的 Java 版本要好。
- ❷ 记录有多少请求产生了成功的匹配。
- ❸ 一次失掉多少个。
- ❹ 我们将在下一个示例中覆盖此方法，以更改用于该度量的关键字。

CounterService 和 GaugeService 在事务中更新捕获指标。为了能够捕捉指标，我们需要更新代码，以便在值得观察的事件发生时发出正确的指标。将检测插入到业务组件的请求路径并不总是很容易，也许追溯捕捉这些指标会更容易一些。Spring Boot Actuator 的 `org.springframework.boot.actuate.endpoint.PublicMetrics` 接口支持集中度量收集。Spring Boot 在内部提供了这个接口的实现，以表示关于 JVM 环境、Apache Tomcat、配置的 DataSource 等的信息。在示例 13-3 中，我们将看到，如何在该应用程序中捕获有关客户的信息。

示例13-3 自定义的PublicMetrics实现，其显示系统中存在多少客户

```
package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.endpoint.PublicMetrics;
import org.springframework.boot.actuate.metrics.Metric;
import org.springframework.stereotype.Component;

import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

@Component
class CustomerPublicMetrics implements PublicMetrics {

    private final CustomerRepository customerRepository;

    @Autowired
    public CustomerPublicMetrics(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    @Override
    public Collection<Metric<?>> metrics() {

        Set<Metric<?>> metrics = new HashSet<>();

        long count = this.customerRepository.count();

        ❶ Metric<Number> customersCountMetric = new Metric<>("customers.count", count);
        metrics.add(customersCountMetric);
        return metrics;
    }
}
```

❶ 此度量报告数据库中 Customer 记录的聚合计数。

目前，我们将度量视为固定时间点的量。它们代表了你查看时的值。这些值是有用的，但它们没有上下文。对于某些值，一个固定的时间点值是毫无意义的。不从历史的角度看，很难知道一个值是意味着改善还是回归。给定时间轴，我们可以取一个值并得出统计数据：平均数、中位数、百分位数等。

Spring Boot Actuator 可以与 Dropwizard Metrics 库无缝集成。Coda Hale (<http://twitter.com/coda>) 在 Yammer 开发了 Dropwizard 度量库，以捕获量表、计数器和其他一些度量

类型。将 `io.dropwizard.metrics:metrics-core` 添加到你的类路径中。

Dropwizard Metrics 库包含对仪表 (meter) 的支持。仪表度量一段时间内的事件发生率 (例如, “每秒订单数”)。如果 Dropwizard Metrics 库位于类路径中, 并且通过 `CounterService` 或 `GaugeService` 捕获的任何度量标准使用前缀 `meter`, 则 Spring Boot Actuator 框架将委托给 Dropwizard Metrics Meter 实现来计算并保持这个度量。

Dropwizard 度量文档 (<https://dropwizard.github.io/metrics/3.1.0/manual/core/>) 中有一段描述: “度量通过几种不同的方式度量事件的速率, 平均速率是事件平均速率, 这通常对琐事很有用。它代表了应用程序整个生命周期的总速率 (例如, 处理的请求总数除以进程运行的秒数), 它不提供最近状态。仪表还记录了三种不同的指数加权移动平均速率: 1、5 和 15 分钟移动平均线。”

Meter 不需要保留它记录的所有值, 因为它使用指数加权移动平均。随着时间的推移保留了一组值。这使得即使在很长一段时间内也能保持内存效率。我们修改示例以使用 Meter, 然后检查记录的度量效果。我们将简单地扩展示例 13-3 中的 `CustomerRestController`, 覆盖度量的 `metricPrefix` 方法, 使用 `meter` 作为所有 `CounterService` 指标的前缀 (参见示例 13-4 和示例 13-5)。

示例13-4 使用CounterService和Dropwizard Metrics库收集度量指标

```
package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.CounterService;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/metered/customers")
public class MeterCustomerRestController extends CustomerRestController {

    @Autowired
    MeterCustomerRestController(CustomerRepository repository,
        CounterService counterService) {
        super(repository, counterService);
    }

    @Override
    protected String metricPrefix(String k) {
        return "meter." + k; ❶
    }
}
```

❶ 使用 `meter` 作为所有记录的度的量的前缀。

示例13-5 由Dropwizard Meter支持的度量标准

```
{
  "meter.customers.read.fifteenMinuteRate" : 0.102683423806518,
  "meter.customers.read.meanRate" : 0.00164167411117908,
  "meter.customers.read.fiveMinuteRate" : 0.0270670566473226,
  "meter.customers.read.oneMinuteRate" : 9.07998595249702e-06
  ...
}
```

计数器 (counter) 和量表 (gauge) 捕获单个值。仪表 (meter) 捕捉一段时间内的数值。仪表不会告诉我们有关数据集中的值的频率的信息。直方图是数据流中值的统计分布：显示某个值出现的次数。它可以回答这样的问题，比如“购物车中有多件物品的订单的百分比？” Dropwizard 指标 Histogram 测量最小值、最大值、平均值、中位数以及百分位数 (第 75、90、95、98、99 和 99.9 百分位数)。

如果学习过基本的统计学课程，那么你应该知道我们需要所有的数据点才能以完美的准确度推导出这些数值。即使在很短的一段时间内，这也可能是一个压倒性的数据量。假设你的应用程序每秒看到 1 000 个逻辑事务，并且每个请求有 10 个请求或操作。一天就有 864 000 000 个数值 ($24 \times 60 \times 60 \times 1000 \times 10$)！如果我们使用的是 Java，那么每个 long 型就有 8 字节，每天的数据量超过 6000 兆字节！大部分应用程序不会有这么多的流量，但有些应用程序会有。无论哪种方式，上下调整它并不难，看到它最终会成为你的一个问题。

Dropwizard 度量库使用水库采样 (*reservoir sampling*) 来保持一个具有代表性的统计样本。随着时间的推移，Dropwizard 直方图创建较旧值的样本，并使用这些样本来推导新的样本。结果并不完美 (有损)，但效率很高。如果 Dropwizard Metrics 库位于 classpath 中，则 Spring Boot Actuator 框架将自动将使用 GaugeService 提交的值中以 histogram. 关键字为前缀的度量转换为 Dropwizard Histogram。

我们来看示例 13-6，它捕获文件上传的直方图。

示例13-6 使用Dropwizard Metrics直方图实现收集文件上传大小的分布

```
package demo.metrics;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.GaugeService;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
```



```
import org.springframework.web.multipart.MultipartFile;

@RestController
@RequestMapping("/histogram/uploads")
public class HistogramFileUploadRestController {

    private final GaugeService gaugeService;

    private Log log = LoggerFactory.getLog(getClass());

    @Autowired
    HistogramFileUploadRestController(GaugeService gaugeService) {
        this.gaugeService = gaugeService;
    }

    @RequestMapping(method = RequestMethod.POST)
    void upload(@RequestParam MultipartFile file) {
        long size = file.getSize();
        this.log.info(String.format("received %s with file size %s",
            file.getOriginalFilename(), size));
        this.gaugeService.submit("histogram.file-uploads.size", size); ❶
    }
}
```

❶ 在所有记录的指标上加上 `histogram.` 前缀，以便在后台将其转换为 Dropwizard Histogram 实例。

示例 13-6 维护了一个文件上传大小的直方图。我们使用 `curl` 随机地上传了三个不同大小的文件（参见表 13-2）。

表13-2

文件大小	文件名	频率
8.0KB	\${HOME}/Desktop/1.png	2
32KB	\${HOME}/Desktop/2.png	5
40KB	\${HOME}/Desktop/3.png	3

`/metrics` 确认数据表所告诉我们的信息（如示例 13-7 所示）。

示例13-7 由Dropwizard直方图支持的度量标准

```
{
...
    "histogram.file-uploads.size.snapshot.98thPercentile" : 38803,
    "histogram.file-uploads.size.snapshot.999thPercentile" : 38803,
    "histogram.file-uploads.size.snapshot.median" : 29929,
    "histogram.file-uploads.size.snapshot.mean" : 27154.1998413605,
    "histogram.file-uploads.size.snapshot.75thPercentile" : 38803,
```

```

    "histogram.file-uploads.size.snapshot.min" : 6347,
    "histogram.file-uploads.size.snapshot.max" : 38803,
    "histogram.file-uploads.size.count" : 10,
    "histogram.file-uploads.size.snapshot.95thPercentile" : 38803,
    "histogram.file-uploads.size.snapshot.99thPercentile" : 38803,
    "histogram.file-uploads.size.snapshot.stdDev" : 11639.4103925448,
    ...
}

```

Dropwizard Metrics 库也支持计时器。计时器测量特定的代码被调用的速度和持续时间的分布。它回答了这样一个问题：“对于一个给定的请求类型，通常会花费多少时间？”什么是非典型的持续时间？Spring Boot Actuator 框架会自动将任何以 `timer.` 开头的度量值转换成 `Timer`，该度量值是通过 `GaugeService` 提交的。

你可以使用各种机制对请求计时。Spring 本身有历史悠久的 `StopWatch` 类，它已经十分完善（如示例 13-8 所示）。

示例13-8 使用Spring `StopWatch`捕获计时

```

package demo.metrics;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.metrics.GaugeService;
import org.springframework.http.ResponseEntity;
import org.springframework.util.StopWatch;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class TimedRestController {

    private final GaugeService gaugeService;

    @Autowired
    public TimedRestController(GaugeService gaugeService) {
        this.gaugeService = gaugeService;
    }

    @RequestMapping(method = RequestMethod.GET, value = "/timer/hello")
    ResponseEntity<?> hello() throws Exception {
        StopWatch sw = new StopWatch(); ❶
        sw.start();
        try {
            Thread.sleep((long) (Math.random() * 60) * 1000);
            return ResponseEntity.ok("Hi, " + System.currentTimeMillis());
        }
        finally {

```



```

    sw.stop();
    this.gaugeService.submit("timer.hello", sw.getLastTaskTimeMillis());
}
}
}

```

❶ 这是 Spring 框架 Stopwatch，我们在这里计算请求花了多长时间。

计时器为我们提供了一个直方图，其包括很多信息，特别是持续时间（见示例 13-9）。

示例13-9 计时器度量输出

```

{
  ...
  "timer.hello.snapshot.stdDev" : 11804,
  "counter.status.200.timer.hello" : 7,
  "timer.hello.snapshot.75thPercentile" : 35004,
  "timer.hello.meanRate" : 0.0561559793104086,
  "timer.hello.snapshot.mean" : 27639,
  "timer.hello.snapshot.min" : 2007,
  "timer.hello.snapshot.max" : 42003,
  "timer.hello.snapshot.median" : 35004,
  "timer.hello.snapshot.98thPercentile" : 42003,
  "timer.hello.fifteenMinuteRate" : 0.182311662062598,
  "timer.hello.snapshot.99thPercentile" : 42003,
  "timer.hello.snapshot.999thPercentile" : 42003,
  "timer.hello.oneMinuteRate" : 0.0741487724647533,
  "timer.hello.fiveMinuteRate" : 0.153231174431025,
  "timer.hello.count" : 7,
  "gauge.response.timer.hello" : 28008,
  "timer.hello.snapshot.95thPercentile" : 42003
  ...
}

```

Dropwizard 指标库丰富了 Spring Boot 度量子系统。它使我们有机会获得一大堆我们原本没有的数据。

指标的连接视图

到目前为止，我们的例子都运行在单个节点或主机上，如一个实例或一个主机。当我们扩大规模时，集中来自所有服务和实例的指标将变得至关重要。我们可以使用时间序列数据库（TSDB）来集中收集和存储指标。时间序列数据库是一种对收集、分析以及度量指标的可视化进行了优化的数据库。有许多流行的时间序列数据库，如 Ganglia (<http://ganglia.info>)、Graphite (<https://github.com/graphite-project/>)、OpenTSDB (<http://opentsdb.net>)、InfluxDB (<https://influxdata.com>) 和 Prometheus (<https://prometheus.io>)。时间序列数据库存储了一段时间内给定键的值。它们通常与支持对时间序列数据库

中的数据绘图的工具协同工作。对时间序列数据进行绘制有许多可以使用的精细的技术，其中最流行的是 Grafana (<http://grafana.org/>)。其他可视化技术比比皆是。许多公司已经开源了他们的可视化工具，比如 Vimeo (<https://github.com/vimeo/graph-explorer>) 的图形浏览器、TicketMaster 的 Metrilyx (<https://tech.ticketmaster.com/2014/08/14/announcing-metrilyx/>) 和 Square's Cubism.js (<http://square.github.io/cubism/>)。

Spring Boot 支持使用 MetricsWriter 接口的实现将度量写入时间序列数据库。开箱即用，Spring Boot 通过 Spring 框架 MessageChannel 或任何支持 StatsD 协议 (<https://github.com/etsy/statsd/wiki>) 的服务，可以将度量发布到 Redis 实例、JMX（可能对开发更有用）。StatsD 是最初 Etsy 中的人使用 Node.js 写的一个代理，作为 Graphite/Carbon 的代理，但是因为该协议本身已经变得非常流行，以至于许多客户端和服务现在都支持这个协议，StatsD 本身还支持多个除了 Graphite 之外的后端实现，如 InfluxDB (<https://influxdata.com>)、OpenTSDB (<http://opentsdb.net>) 和 Ganglia (<http://ganglia.info>)。为了利用各种 MetricWriter 实现，只需定义一个适当类型的 bean 并用 @ExportMetricWriter 对其进行注释即可。如果 StatsD 不适合你的用例或者你想要做更多的控制，Dropwizard 还支持通过其 *Reporter 实现向下游系统发布度量指标。

度量数据的维度

你在时间序列数据库中创建的数据仍然是数据，即使该数据的维度非常有限：度量中至少包含一个键和一个值。这就存在一些问题，*schema* 的方式很少。不同的时间序列数据库对此做了改进并提供其他维度的数据。有些提供 *label* 或 *tag* 的概念。但是，所有时间序列数据库都有一个共同点，就是只有一个键。我们看到的大多数实现都使用分层键（例如 a.b.c）。许多时间序列数据库支持 glob 查询（例如 a.b.*），这将返回与前缀 a.b 相匹配的所有度量。键中的每个路径组件都应该有一个明确定义的目的，并且经常改变的路径组件应尽可能位于层次结构的深层中。设计你的键和度量指标，以支持越来越多的指标。通过分级键或其他维度，如 *label* 和 *tag*，仔细考虑你在指标中捕捉到的内容。

如何在同一时间序列数据库中编码对不同产品的请求？捕获组件名称，例如 *order-service*。

如何编码不同类型的活动或流程？HTTP 请求、基于消息传递的后台请求、后台批量作业还是其他？*order-service.tasks.fulfillment.validate-shipping* 或 *order-service.requests.new-order*？

考虑如何对信息进行编码，使其最终与面向产品管理的系统（如 HP 的 Vertica，<https://www.vertica.com/>，或 Facebook 的 Scuba）相关联？虽然我们想将所有运维监控直接转化为业务指标，但事实并非如此。尽管如此，有一种方法可以捕获这些信息并将它们连接起来。你可以使用度量的关键字本身或 *label* 和 *tag* 预先做到这一点。



如何将请求与 A/B 测试（或实验）关联起来，在这些测试中，群体样本通过运行特定路径下的代码，所产生的行为与大多数请求不同。这有助于判断某个功能是否有效并作出评测。也就是说，你可能在两个不同的代码路径中有相同的度量标准，其中一个实验用的。许多组织都有用于实验的系统，以及应该纳入指标的实验室数据。

模式（schema）设计是一个主观的事情，时间序列数据库在收集的数据的丰富性和规模上不尽相同！有些时间序列数据库是有损的，而有些时间序列数据库是可以横向扩展的，容量几乎是无限的。你应该像选择其他数据库一样选择你的时间序列数据库，并仔细考虑模式设计。一方面，开发人员应该很容易无成本地捕捉指标。另一方面，多一些思考对于以后的监控会有好处。

Spring Boot 应用程序传送度量指标

你可以随时找到许多时间序列数据库的托管版本。一个是 Hosted Graphite (<https://www.hostedgraphite.com/>)，其很容易整合。它预先配置有 Graphite、Graphite Composer 和 Grafana。Grafana 和 Graphite Composer 都可以让你根据收集的指标构建图表。当然，你也可以运行自己的 Graphite 实例，但请记住，我们的目标是尽快地投入生产，因此我们倾向于选择基于云的服务（软件即服务，即 SaaS）；除非我们可以出售它，否则我们通常不独立运行软件。Spring Boot 开发人员在连接 Graphite 时有几个可以选择的选项。你可以使用 Spring Boot 的 StatsdMetricWriter，这个 StatsdMetricWriter 使用的是 StatsD 协议，可以和许多前面提到的后端一起工作。除了 StatsD 之外，你还可以使用无数的原生 Dropwizard Metrics reporter 实现，例如，你可能更喜欢使用原生协议。在示例 13-10 中演示了如何与 Graphite 进行通信。

示例13-10 配置GraphiteReporter实例的Dropwizard度量

```
package demo.metrics;

import com.codahale.metrics.MetricRegistry;
import com.codahale.metrics.graphite.Graphite;
import com.codahale.metrics.graphite.GraphiteReporter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.annotation.PostConstruct;
import java.util.concurrent.TimeUnit;

@Configuration
class ActuatorConfiguration {

    ActuatorConfiguration() {
        java.security.Security.setProperty("networkaddress.cache.ttl", "60"); ❶
    }
}
```



```

}

@Bean
GraphiteReporter graphiteWriter(
    @Value("${hostedGraphite.apiKey}") String apiKey, ❷
    @Value("${hostedGraphite.url}") String host,
    @Value("${hostedGraphite.port}") int port, MetricRegistry registry) {

    GraphiteReporter reporter = GraphiteReporter.forRegistry(registry)
        .prefixedWith(apiKey)
        .build(new Graphite(host, port));
    reporter.start(1, TimeUnit.SECONDS);
    return reporter;
}
}

```

- ❶ 阻止 DNS 缓存，因为 HostedGraphite.com 节点可能会迁移并映射到新的 DNS 路由。
- ❷ HostedGraphite.com 的服务 (<http://HostedGraphite.com>) 使用 API 密钥建立认证，并期望你将其作为 prefix 字段的一部分进行传输。诚然，这是一个很丑陋的做法，但没有其他明显的地方放置认证信息！

将流量导到 CustomerRestController 或 MeterCustomerRestController，你将看到图形中反映的流量，你可以在 Grafana 界面或 Graphite 编辑器界面的 HostedGraphite.com 上创建图形，如图 13-1 和图 13-2 所示。

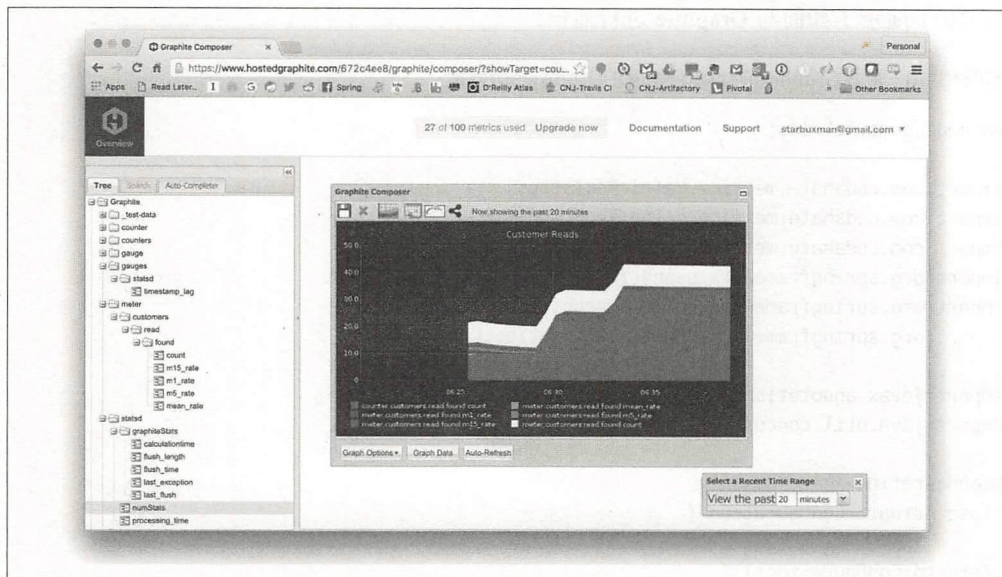


图13-1 Graphite Composer仪表板



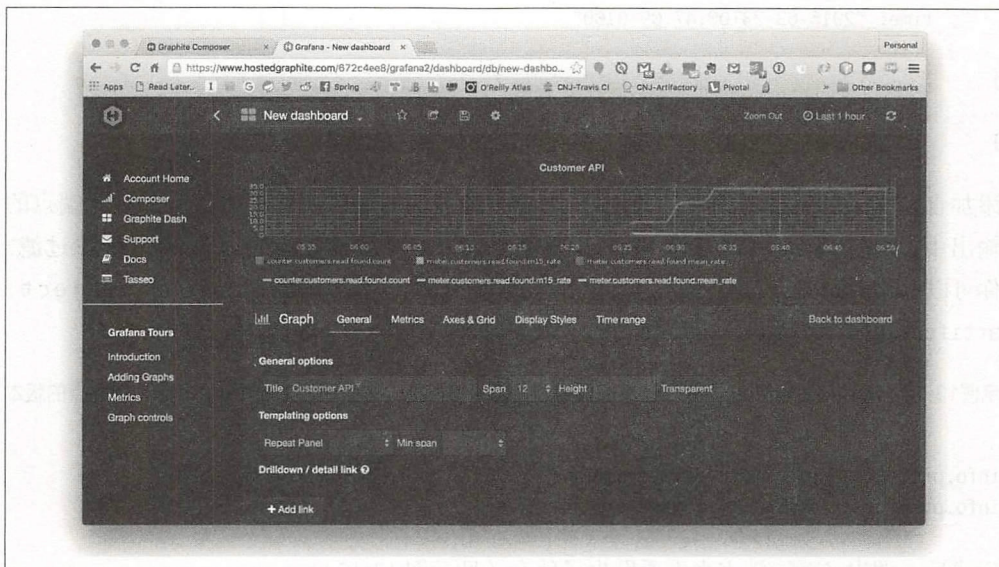


图13-2 Grafana仪表板

通过 /info 端点识别服务

在理想情况下，你会在持续交付管道中开发代码。在持续交付管道中，每个提交都可能被推送到生产。在理想情况下，一天内有多次提交。如果出现问题，人们首先想知道的是代码运行的是哪个版本。你可以使用 /info 端点来识别服务。

/info 端点有意留空。/info 端点是放置服务本身信息的地方。服务名称是什么？哪个 git commit 触发最终导致生产的构建？服务版本是什么？

你可以在环境中通过正常通道（application.properties、application.yml 等）中的 info. 前缀属性来设置自定义属性。你也可以通过 pl.project13.maven:git-commit-id-plugin Maven 插件添加 git source 代码仓库状态信息。这个插件是为 Spring Boot 父级 Maven 构建中的 Maven 用户预配置的。它生成一个包含 git.branch 和 git.commit 属性的 git.properties 文件。如果其可用，/info 端点将知道如何查找它（见示例 13-11）。

示例13-11 从/info端点暴露的Git分支、提交的ID和时间

```
{
  git: {
    branch: "master",
    commit: {
      id: "407359e",
```



```

        time: "2016-03-23T00:47:09+0100"
    }
}
...
}

```

添加自定义属性也非常简单。任何以 `info.` 为前缀的环境属性都将被添加到此端点的输出中。Spring Boot 的默认 Maven 插件配置，已经被设置为处理 Maven 资源过滤。你可以利用 Maven 资源过滤来发布在构建时捕获的自定义属性，如 `Maven project.artifactId` 和 `project.version`（见示例 13-12）。

示例13-12 通过在使用Maven资源过滤的构建过程中设置属性，捕获项目的`artifactId`和`/info`端点的版本等自定义构建时信息

```

info.project.version=@project.version@
info.project.artifactId=@project.artifactId@

```

完成后，调出 `/info` 端点来看看发生了什么（见示例 13-13）。

示例13-13 使用`/info`端点捕获项目的`artifactId`和版本等自定义构建时信息

```

{ ...
  project: {
    artifactId: "actuator",
    version: "1.0.0-SNAPSHOT"
  }
}

```

健康检查

应用程序需要一种基础设施的健康检查方法。良好的健康检查应该提供一个综合状态，总结报告各个组件的状态。负载均衡器经常使用健康检查来确定节点的可用性。负载均衡器可能会根据返回的 HTTP 状态码来驱逐节点。 `org.springframework.boot.actuate.endpoint.HealthEndpoint` 收集应用上下文中的所有 `org.springframework.boot.actuate.health.HealthIndicator` 实现并公开它们。示例 13-14 显示了我们的应用程序中默认 `/health` 端点的输出。

示例13-14 应用程序的默认`/health`端点的输出

```

{
  status: "UP",
  diskSpace: {
    status: "UP",
    total: 999334871040,

```




```

        free: 735556071424,
        threshold: 10485760
    }, redis: {
        status: "UP",
        version: "3.0.7"
    },
    db: {
        status: "UP",
        database: "H2",
        hello: 1
    }
}

```

Spring Boot 根据 JavaMail、MongoDB、Cassandra、JDBC、SOLR、Redis、ElasticSearch、文件系统等的各种自动配置自动注册常用的 HealthIndicator 实现。这些健康指标独立于服务，是关于你的服务工作的信息，可能会失败。在上面的例子中，我们看到 Redis、文件系统和 JDBC DataSource 都被自动计算了。

下面我们来设置一个自定义的 HealthIndicator。HealthIndicator 的协议很简单：当被询问时，返回一个具有适当状态的 Health 实例。系统中的其他组件需要能够影响返回的 Health 对象。你可以直接注入相关的 HealthIndicator 并在每个可能影响该状态的组件中操作其状态，但是这会很多应用程序代码耦合到次要问题（健康状态）中。另一种方法是使用 Spring 的 ApplicationContext 事件总线来发布组件中的事件，并根据已确认的事件操作 HealthIndicator。

在下面的代码块中，我们将建立一个情绪健康指示器（见示例 13-15），当它收到一个 SadEvent（见示例 13-16）或一个 HappyEvent（见示例 13-17）时显示高兴（UP）或者悲伤（DOWN）的情绪。

示例13-15 情绪的HealthIndicator

```

package demo.health;

import org.springframework.boot.actuate.health.AbstractHealthIndicator;
import org.springframework.boot.actuate.health.Health;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.Optional;

@Component
class EmotionalHealthIndicator extends AbstractHealthIndicator {

```



```
private EmotionalEvent event;
```

```
private Date when;
```

❶

```
@EventListener
```

```
public void onHealthEvent(EmotionalEvent event) {  
    this.event = event;  
    this.when = new Date();  
}
```

❷

```
@Override
```

```
protected void doHealthCheck(Health.Builder builder) throws Exception {  
    // @formatter:off
```

```
    Optional
```

```
        .ofNullable(this.event)
```

```
        .ifPresent(  
            evt -> {
```

```
                Class<? extends EmotionalEvent> eventClass = this.event.getClass();
```

```
                Health.Builder healthBuilder = eventClass
```

```
                    .isAssignableFrom(SadEvent.class) ? builder
```

```
                    .down() : builder.up();
```

```
                String eventTimeAsString = this.when.toInstant().toString();
```

```
                healthBuilder.withDetail("class", eventClass).withDetail("when",  
                    eventTimeAsString);
```

```
            });
```

```
    // @formatter:off
```

```
}
```

```
}
```

❶ 我们使用 `@EventListener` 注解将此侦听器方法连接到 `ApplicationContext` 事件。

❷ `doHealthCheck` 方法使用 `Health.Builder` 根据最近记录的已知 `EmotionalEvent` 切换健康指示器的状态。

示例13-16 SadEvent

```
package demo.health;
```

```
public class SadEvent extends EmotionalEvent {  
}
```

示例13-17 HappyEvent

```
package demo.health;
```




```
public class HappyEvent extends EmotionalEvent {  
}
```

现在，ApplicationContext 中的任何组件只需要发布一个适当的事件来触发相应的状态改变（见示例 13-18）即可。

示例13-18 情绪的REST端点

```
package demo.health;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.ApplicationEventPublisher;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
class EmotionalRestController {  
  
    private final ApplicationEventPublisher publisher;  
  
    @Autowired  
    EmotionalRestController(ApplicationEventPublisher publisher) { ❶  
        this.publisher = publisher;  
    }  
  
    @RequestMapping("/event/happy")  
    void eventHappy() {  
        this.publisher.publishEvent(new HappyEvent()); ❷  
    }  
  
    @RequestMapping("/event/sad")  
    void eventSad() {  
        this.publisher.publishEvent(new SadEvent());  
    }  
}
```

- ❶ Spring 会自动公开 ApplicationEventPublisher 接口的实现，以便在组件代码中使用。事实上，Spring 正在用来运行应用程序的 ApplicationContext 可能已经是 ApplicationEventPublisher。
- ❷ 分派组件之间的事件。

通过事件可以更轻松地支持操作信息，而不用在非业务逻辑上消耗过多精力，例如，运行健康状况端点。



审计事件

事件是捕捉系统中几乎所有东西的好方法。Spring Boot 支持使用事件来做审计，审计通过审计事件将应用程序中的事件与触发它们的已认证用户绑定。我们来看一下在类路径中还拥有 Spring Security (org.springframework.boot: spring-boot-starter-security) 的 REST 端点。为了做一个简单的演示程序，我们配置了一个自定义的 UserDetailsService 实现 (见示例 13-19)。

示例13-19 一些硬编码的用户

```
package com.example;

import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.Arrays;
import java.util.Optional;
import java.util.Set;
import java.util.concurrent.ConcurrentSkipListSet;

@Service
class SimpleUserDetailsService implements UserDetailsService {

    private final Set<String> users = new ConcurrentSkipListSet<>();

    SimpleUserDetailsService() {
        ❶
        this.users.addAll(Arrays.asList("pwebb", "dsyer", "mbhave", "snicoll",
            "awilkinson"));
    }

    @Override
    public UserDetails loadUserByUsername(String s)
        throws UsernameNotFoundException {
        ❷
        return Optional
            .ofNullable(this.users.contains(s) ? s : null)
            .map(x -> new User(x, "pw", AuthorityUtils.createAuthorityList("ROLE_USER")))
            .orElseThrow(() -> new UsernameNotFoundException("couldn't find " + s + "!!"));
    }
}
```


❶ 硬编码用户列表（dsyer、pwebb 等）。

❷ 密码（每个用户的 pw——请勿模仿！）和一个固定角色（ROLE_USER）。

Spring Boot 自动配置默认情况下使用 HTTP BASIC 身份验证锁定 HTTP 端点。Spring Security 将生成与身份验证和授权相关的事件：是否有人进行了身份验证（或尝试失败），是否有人退出等。你还可以创建自己的审计事件。我们来看一个简单的 HTTP 端点示例，它依赖于 Spring Security 将当前已验证的 `java.security.Principal` 注入 Spring MVC 处理程序方法（见示例 13-20）中。

示例13-20 一个简单（但安全）的HTTP端点

```
package com.example;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.audit.AuditEvent;
import org.springframework.boot.actuate.audit.AuditEventRepository;
import org.springframework.boot.actuate.audit.listener.AuditApplicationEvent;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import java.security.Principal;
import java.util.Collections;
```

```
@RestController
```

```
class GreetingsRestController {
```

```
    public static final String GREETING_EVENT = "greeting_event".toUpperCase();
```

```
    private final ApplicationEventPublisher appEventPublisher;
```

```
    @Autowired
```

```
    GreetingsRestController(ApplicationEventPublisher appEventPublisher) {
```

```
        this.appEventPublisher = appEventPublisher;
```

```
    }
```

```
    @GetMapping("/hi")
```

```
    String greet(Principal p) { ❶
```

```
        String msg = "hello, " + p.getName() + "!";
```

```
        AuditEvent auditEvent = new AuditEvent(p.getName(), ❷
```

```
            GREETING_EVENT, ❸
```

```
            Collections.singletonMap("greeting", msg)); ❹
```

```
        this.appEventPublisher.publishEvent( ❺
```

```

        new AuditApplicationEvent(auditEvent));

    return msg;
}
}

```

- ❶ 注入当前认证的 Principal。
- ❷ 使用它来创建一个 AuditEvent，取消引用已认证的 Principal 名字。
- ❸ 以及一个事件名称（这是任意的，对你的系统有意义的）。
- ❹ 你希望包含在日志中的任何额外的元数据。
- ❺ 最后，使用 Spring 应用程序上下文事件机制，并使用一个包装程序 AuditApplicationEvent 来分派 AuditEvent 事件。

你可以调用安全端点，使用 HTTP BASIC 进行身份验证。这里使用友好的 httpie 客户端，但是你也可以使用任何其他的客户端（见示例 13-21）。

示例13-21 /auditevents执行器HTTP端点的输出

```

{
  "events" : [
    {
      "timestamp" : "2017-04-26T14:01:10+0000",
      "principal" : "dsyer",
      "type" : "AUTHENTICATION_SUCCESS",
      "data" : {
        "details" : {
          "sessionId" : null,
          "remoteAddress" : "127.0.0.1"
        }
      }
    },
    {
      "timestamp" : "2017-04-26T14:01:10+0000",
      "data" : {
        "greeting" : "hello, dsyer!"
      },
      "type" : "GREETING_EVENT",
      "principal" : "dsyer"
    }
  ]
}

```

审计事件机制使捕获有关系统中用户的信息变得十分简单。Spring Boot 有一个名为 AuditListener 的组件，用于监听事件并使用 AuditEventRepository 实现记录它们（见

示例 13-22)。在默认情况下，这个实现是驻于内存中的，但是使用某种持久的后台存储来实现也容易。也可以贡献你自己的实现，Spring Boot 将会十分感谢你的贡献。

你也可以使用其他 Spring 事件侦听器机制来侦听（和响应）自己代码中的审计事件。

示例13-22 一个简单的AuditApplicationEvent监听器

```
package com.example;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.actuate.audit.AuditEvent;
import org.springframework.boot.actuate.audit.listener.AbstractAuditListener;
import org.springframework.boot.actuate.audit.listener.AuditApplicationEvent;
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
class SimpleAuditEventListener {

    private Log log = LogFactory.getLog(getClass());

    @EventListener(AuditApplicationEvent.class)
    public void onAuditEvent(AuditApplicationEvent event) {
        this.log.info("audit-event: " + event.toString());
    }
}
```

应用程序日志

现在我们有自动驾驶汽车和智能家居可以与交互。我们可以瞬间启动一千台服务器！然而，可敬的日志文件仍然是我们理解特定节点或系统行为的最好方法之一。日志反映了过程的节奏和活动。日志记录要求在代码中加入侵入性语句，但是生成的日志文件本身是与系统分离的。我们有完整的完全围绕着从日志分析到数据挖掘的生态系统、工具、语言和大数据平台。

关于日志你需要考虑下面两个问题。

日志输出

将日志输出到哪里？输出到文件里？在控制台上？在 SyslogD 服务中？

日志级别

什么粒度的输出？你想要打印出每一步小小的操作，还是只想要威胁系统的事情？

指定日志输出

Spring Boot 应用程序中的日志默认输出到控制台上。你可以配置输出到文件或其他日志附加器中，但控制台是一个合理的默认输出位置。

在开发时，你希望能够实时查看日志，如果你在云环境中运行应用程序，则不必担心日志的路由位置。这是十二要素宣言的原则之一：

十二要素应用程序从不关心其输出流的路由或存储。应用程序不应该尝试写入或管理日志文件。相反，所有正在运行的进程都将其未缓冲的事件流写入 stdout。在本地开发期间，开发人员将在终端的前台查看此流，观察应用程序的行为。

日志收集器或日志复用器，如 Cloud Foundry 的 Loggregator (<https://github.com/cloudfoundry/loggregator>) 或 Logstash，从不同的进程生成日志，并将它们统一为单个流，可能将该流转发到某个位置进行分析。日志数据应尽可能结构化——使用一致的分隔符、定义组，并支持类似 log schema 的东西来支持分析。日志应该被视为事件流，它们描述了这个系统的行为。日志信息可能是一个进程的输出和另一个下游分析进程的输入。Logstash 是一个非常受欢迎的日志多路复用器，它提供了众多的插件，让你能够管理多个输入源的日志，并将这些日志连接到像 Elasticsearch 这样的集中分析系统，这是一个基于 Lucene 的全文搜索引擎。

还有一个日志复用器 Loggregator 会使用 `cf logs $YOUR_APP_NAME` 或任何 SyslogD 协议兼容的服务，包括 ElasticSearch (通过 Logstash)、Papertrail (<https://papertrailapp.com>)、Splunk (<http://www.splunk.com>)、Splunk Storm、SumoLogic，当然还有 SyslogD 本身 (<http://bit.ly/2sabkxc>)。像使用用户提供的服务一样配置一个日志汇集，使用 `-l` 来表明它是日志汇集，如示例 13-23 所示。

示例13-23 创建一个用户提供的服务

```
cf cups my-logs -l syslog://logs.papertrailapp.com:PORT
```

这只是一个可供任何应用程序绑定的服务，如示例 13-24 所示。

示例13-24 将服务绑定到应用程序

```
cf bind-service my-app my-logs && cf restart my-app
```

Loggregator 还通过 websocket 协议发布日志消息，所以通过编程监听来自任何 Cloud Foundry 应用程序的日志也非常简单。我们使用的是 Java，所以 Cloud Foundry Java 客户端可以轻松地整合这个 websocket feed。

Pivotal Cloud Foundry 还提供相关日志记录 (如图 13-3 所示)，它会在时间线上显示请

求的度量，然后展示指定时间段内的日志。

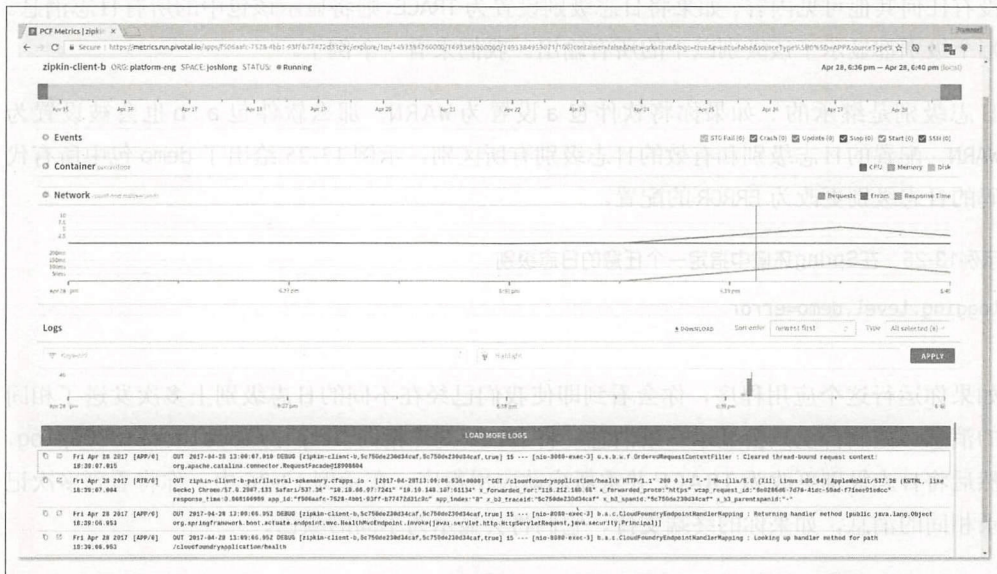


图13-3 Pivotal AppManager上的相关日志记录

指定日志级别

日志已成为应用程序状态的自然延伸，在你开始选择使用哪种日志记录技术的时候，可能觉得无所适从，感到困惑。如果你使用的是 Spring Boot，那么使用默认的就是可以。Spring Boot 使用 Commons Logging 进行所有内部日志记录，但是将底层日志实现开放。它为 JDK 的日志记录支持 Log4J、Log4J2 和 Logback 提供了默认配置。在各种情况下，记录器都预先被配置为使用控制台输出，也可选择文件输出。

在默认情况下，Spring Boot 将使用 Logback，它可以捕获和转发其他日志记录技术生成的日志，如 Apache Commons Logging、Log4j、Log4J2 等。这是什么意思？这意味着所有在 classpath 下的与日志生成有关的依赖将工作得很好，并以一个众所周知的配置格式到达控制台，包括日期和时间、日志级别、进程 ID、线程名称、记录器名称和实际的日志消息。如果你在控制台上查看日志，可以看到，它甚至会包含颜色代码！

通过在 Spring 环境中指定级别（作为 application.properties 或 Spring Cloud Config Server 实例中的属性），可以使用 Spring Boot 来统一管理日志级别。Spring 可以理解并适当地将它们映射到底层的日志记录提供程序。有如下所示的日志级别：ERROR、WARN、INFO、DEBUG 或 TRACE。你也可以通过指定 OFF 来关闭所有输出。日志级别按优先级排序。能够看到 DEBUG 中的语句对于开发者的帮助比看到 ERROR 中威胁到稳定性的消息记

录更重要。如果将日志级别设置为 `ERROR`，则除了记录 `ERROR` 的消息之外，该程序包中没有任何其他可见内容。如果将日志级别设置为 `TRACE`，则将显示该包中的所有日志消息。每个级别都显示了该级别以下的所有输出。我们来看一个例子。

日志级别是继承的：如果你将软件包 `a` 设置为 `WARN`，那么软件包 `a.b` 也会被设置为 `WARN`。配置的日志级别和有效的日志级别有所区别。示例 13-25 给出了 `demo` 包中所有代码的日志级别更改为 `ERROR` 的配置。

示例13-25 在Spring环境中指定一个任意的日志级别

```
logging.level.demo=error
```

如果你运行这个应用程序，你会看到即使我们已经在不同的日志级别上多次发送了相同的消息，控制台上只会显示一条消息。将 `HTTP GET` 指向 `http://localhost:8080/log`，然后将日志级别更改为 `TRACE` 并重新启动应用程序。在示例 13-26 中，你将看到多次记录相同的消息，如果你的终端支持，它们将会显示为不同的颜色。

示例13-26 在不同日志级别记录三条消息的Java应用程序

```
package demo;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import javax.annotation.PostConstruct;
import java.util.Optional;

@SpringBootApplication
@RestController
public class LoggingApplication {

    private Log log = LogFactory.getLog(getClass());

    public static void main(String args[]) {
        SpringApplication.run(LoggingApplication.class, args);
    }
}
```



```

LoggingApplication() {
    triggerLog(Optional.empty());
}

@GetMapping("/log")
public void triggerLog(@RequestParam Optional<String> name) {
    String greeting = "Hello, " + name.orElse("World") + "!";
    this.log.warn("WARN: " + greeting); ❶
    this.log.info("INFO: " + greeting);
    this.log.debug("DEBUG: " + greeting);
    this.log.error("ERROR: " + greeting);
}
}

```

❶ 根据你指定的日志级别，将不显示、显示一条或全部日志消息。

刚刚，我们重新启动了进程来查看更新的日志级别。但是我们也可以在进程运行时询问和动态地使用 Spring Boot Actuator /loggers 端点来重新配置日志级别。如果使用 HTTP GET，那么端点会显示应用程序中所有配置的日志级别（见示例 13-27）。

示例13-27 枚举所有日志级别

```

{
  "loggers" : {
    ... ❶
    "org.springframework.boot.actuate.endpoint" : {
      "effectiveLevel" : "INFO",
      "configuredLevel" : null
    },
    "demo" : {
      "effectiveLevel" : "ERROR",
      "configuredLevel" : "ERROR"
    }
  },
  "levels" : [
    "OFF",
    "ERROR",
    "WARN",
    "INFO",
    "DEBUG",
    "TRACE" ]
}

```

❶ 该示例只摘录了上千行配置中的几行！

你可以使用 /loggers/{logger} 来获取特定记录器的详细信息，其中 {logger} 是你的包或日志层次结构的名称。在我们的例子中，我们可以调用 /loggers/demo 来确认这个特

定级别的配置。你可以调用 `/loggers/ROOT` 来查找通知所有其他未指定和更具体的日志级别的根日志级别。

你还可以向相关日志记录器端点发送 HTTP POST 请求更新日志级别。

示例 13-28 更新 demo 包配置的日志级别。

示例13-28 更新日志级别

```
curl -i -X POST -H 'Content-Type: application/json' \
  -d '{"configuredLevel": "TRACE"}' \
  http://localhost:8080/loggers/demo
```

这非常有用，但只适用于单个实例。如果在云上有多个实例同时运行，那么为部署的应用程序启动或停止日志级别会更有用。如果你使用 Pivotal Web Services 或 Pivotal Cloud Foundry，这很简单。在图 13-4 和图 13-5 中，我们将在 Pivotal AppsManager 仪表板上仔细阅读应用程序的日志，然后重新配置 Spring Boot 应用程序的日志级别。

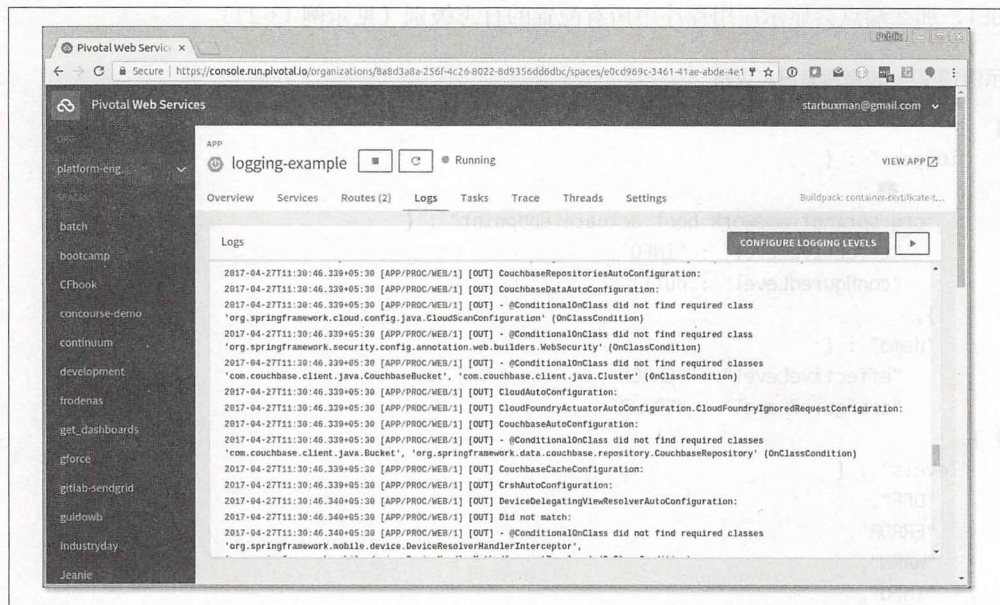


图13-4 从AppsManager控制面板查看应用程序的日志

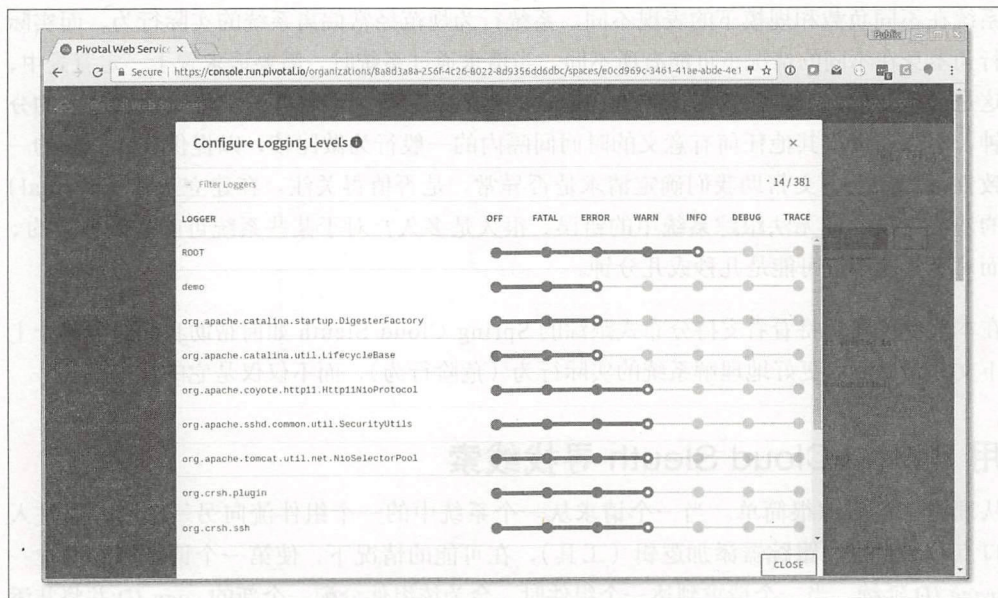


图13-5 在Pivotal Cloud Foundry或Pivotal Web Services上重新配置
Spring Boot应用程序的日志级别

分布式跟踪

有许多选项可以帮助你理解应用程序和性能配置文件。有基于代理的监控技术，像 New Relic，其可以与 Pivotal Cloud Foundry (<https://newrelic.com/partner/pivotal>) 无缝集成；App Dynamics，它也可以与 Pivotal Cloud Foundry (<https://docs.pivotal.io/partners/appdynamics/index.html>) 无缝集成。它们使用 Java 代理和自动检测工具为你提供应用程序性能行为的底层视图。这些工具值得研究，因为它们可以让你以运行时透视的方式查看应用程序的性能。APM 工具可以为你提供跨应用程序端到端行为的、跨语言和技术、从 HTTP 请求到低级数据源访问的仪表板。

云计算技术方面的进步使得构建和部署服务变得更加容易。云计算使我们能够自动消除与提供新服务相关的痛苦。速度的提高反过来又使我们变得更加灵活，思考更小批量的可独立部署的服务。新服务的激增使系统范围和特定于请求的性能特性的推理复杂化。

当一个应用程序的所有功能都存放在一个独立巨石 (monolith) 的应用程序中时，我们把应用程序称为一个大而全，可以像 .war 或者 .ear 一样部署的应用程序。有没有内存泄漏？发生在巨石应用中。组件没有正确处理请求吗？发生在巨石应用中。消息正在丢失？也可能发生在巨石应用中。分解改变了这一切。

系统在不同负载和规模下的表现不同。系统行为规范经常偏离系统的实际行为，而实际行为本身在不同的情况下可能有所不同。当请求通过系统时，需要请求置于一定背景中，这是很重要的。我们需要考虑到特定请求的性质及其行为，将其与类似请求在过去的分钟、小时、天或其他任何有意义的时间间隔内的一般行为做比较，以提供统计学上的一致性抽样。上下文帮助我们确定请求是否异常，是否值得关注。在建立正常（normal）的基线之前，你无法跟踪系统中的错误。很久是多久？对于某些系统可能是微秒级的；而对于其他系统可能是几秒或几分钟。

在本节中，我们将看看支持分布式跟踪的 Spring Cloud Sleuth 如何帮助我们建立这个上下文并帮助我们更好地理解系统的实际行为（危险行为），而不仅仅是它的特定行为。

用 Spring Cloud Sleuth 寻找线索

从理论上说跟踪很简单。当一个请求从一个系统中的一个组件流向另一个时，通过入口点和出口点，跟踪器添加逻辑（工具），在可能的情况下，使第一个请求产生的唯一 *trace ID* 延续。当一个请求到达一个组件时，会为该组件分配一个新的 *span ID* 并将其添加到该跟踪中。跟踪（trace）代表一个请求的整个过程，跨度（span）是在一个过程中的每个单独的跳跃或请求。span 可能包含 tag 或元数据，可用于稍后处理请求，可以将请求与特定事务相关联。span 通常包含通用标签，如开始时间戳和停止时间戳，虽然很容易将语义相关标签（如业务实体标识）与 span 相关联。

假设我们有两个服务，*service-a* 和 *service-b*。如果一个 HTTP 请求到达 *service-a* 后，又通过 Apache Kafka 向 *service-b* 发送了一个消息，那么我们将获得一个 trace ID，但是有两个 span。每个 span 将具有特定请求的标签。第一个 span 可能有 HTTP 请求的细节。第二个 span 可能包含发送给 Apache Kafka 代理的消息的详细信息。

Spring Cloud Sleuth (<http://cloud.spring.io/spring-cloud-sleuth/>) (org.springframework.cloud:spring-cloud-starter-sleuth) 自动地处理常见的通信渠道：

- Apache Kafka 或 RabbitMQ，或任何其他具有 Spring Cloud Stream (<http://bit.ly/2s9PsBY>) binder 的消息系统
- 在 Spring MVC 控制器收到的 HTTP 头
- 通过 Netflix Zuul 微代理的请求
- 使用 RestTemplate 创建的请求
- 通过 Netflix Feign REST 客户端创建的请求
- 大多数其他类型的典型的 Spring 生态系统应用程序可能会遇到的请求和答复

Spring Cloud Sleuth 会为你设置有用的日志格式，记录 trace ID 和 span ID。假设你在

`spring.application.name` 为 `my-service-id` 的微服务中运行启用 Spring Cloud Sleuth 的代码，你会在微服务的日志中看到示例 13-29 所示的日志。

示例 13-29 来自 Spring Cloud Sleuth-instrumented 应用程序的日志

```
2016-02-11 17:12:45.404 INFO [my-service-id,73b62c0f90d11e06,73b6etydf90d11e06,false]
85184 --- [nio-8080-exec-1] com.example.MySimpleComponentMakingARequest : ...
```

在这个例子中，`my-service-id` 是 `spring.application.name`，`73b62c0f90d11e06` 是 `trace ID`，`73b6etydf90d11e06` 是 `span ID`。这些信息非常有用，你可以使用你掌握的任何日志分析工具来挖掘它。如果将所有日志和跟踪信息放在一个可用来查询和分析的位置，则可以看到通过不同服务的请求流。

Spring Cloud Sleuth 工具通常包含两个组件：一个是用于执行某个子系统的 `tracing` 的对象；另一个是该子系统的特定的 `SpanInjector <T>` 实例。跟踪器通常是某种拦截器、侦听器、过滤器等，你可以将其插入跟踪组件的请求流中。如果出于某种原因，你所需要的组件尚未被开箱即用，你也可以创建并贡献你自己的 `tracing` 对象。

多少数据是足够的

应跟踪哪些请求？在理想情况下，我们需要有足够的数据才能看出实时业务流量的趋势。不过，你肯定不想压垮自己的日志和分析基础设施。有些组织可能每一千个请求，或者每十个、每一百万个请求才分析一次！在默认情况下，阈值为 10%，即 0.1，但可以通过配置一个抽样百分比来覆盖该阈值（见示例 13-30）。

示例 13-30 改变抽样阈值百分比

```
spring.sleuth.sampler.percentage = 0.2
```

或者，你可以注册自己的 `Sampler bean` 定义，并决定应该采样哪些请求。你可以做出更明确的选择，例如忽略成功的请求，或者检查某个组件是否处于错误状态。示例 13-31 显示了 `Sampler` 的定义。

示例 13-31 Spring Cloud Sampler 接口

```
package org.springframework.cloud.sleuth;

import org.springframework.cloud.sleuth.Span;

public interface Sampler {
    boolean isSampled(Span s);
}
```

作为参数给出的 `Span` 表示在较大跟踪中当前正在执行的请求的跨度。如果你愿意，你可

以做一些有趣的、特定的请求类型的抽样。例如,可以仅采样 HTTP 状态码为 500 的请求。

确保为你的应用程序和基础设施设定切合实际的期望。可能你的应用程序使用的模式擅长于检测趋势和模式,或者不擅长。这是为了在线遥测。大多数组织不会存储这些数据超过几天,或者上限为一周。

OpenZipkin : 一张图片胜过千丝万缕

数据收集只是一个开始,我们的目标是了解数据,而不仅仅是收集数据。为了掌握全局,我们需要超越单个事件。我们将使用 OpenZipkin 项目 (<https://github.com/openzipkin>)。OpenZipkin 是 Zipkin 的开源版本(见图 13-6)。这个项目起源于 2010 年的 Twitter,并且基于 Google Dapper 论文 (<http://research.google.com/pubs/pub36356.html>)。



图13-6 OpenZipkin是Zipkin的开源版本



以前, Zipkin 的开源版本的发展速度与 Twitter 内部使用的版本不同。OpenZipkin 代表了在这上面的同步努力: OpenZipkin (<https://github.com/openzipkin>) 是 Zipkin, 当我们在本书中引用 Zipkin 时, 我们指的是 OpenZipkin 的版本。

Zipkin 提供客户端可以直接使用的 REST API。这个 REST API 是用 Spring MVC 和 Spring Boot 编写的。Zipkin 甚至支持基于 Spring Boot 的 REST API 实现。使用它就像直接使用 Zipkin 的 `@EnableZipkinServer` 一样简单。Zipkin 服务器通过一个 `SpanStore` 委托写入持久层。目前, 支持使用 MySQL 或者开箱即用的内存 `SpanStore`。

作为与 Zipkin REST API 直接交互的替代方法, 我们也可以通过像 RabbitMQ 或 Apache Kafka 这样的 Spring Cloud Stream 绑定器将消息发布到 Zipkin 服务器上, 这就是你将在示例 13-32 中看到的内容。创建一个新的 Spring Boot 应用程序, 在类路径中添加 `org.springframework.cloud:spring-cloud-sleuth-zipkin-stream`, 然后在 Spring Boot 应

用程序中添加 `@EnableZipkinStreamServer` 来接受和适配传入的 Spring Cloud 基于流的 SleuthSpan 实例到 Zipkin 的 Span 类型中。然后，其使用配置的 SpanStore 持久化它们。你可以使用任何你喜欢的 Spring Cloud Stream 绑定，但在这种情况下，我们将使用 Spring Cloud Stream RabbitMQ (`org.springframework.cloud:spring-cloud-starter-stream-rabbitmq`)。

示例13-32 Zipkin服务器代码

package demo;

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.sleuth.zipkin.stream.EnableZipkinStreamServer;
```

❶

`@EnableZipkinStreamServer`

`@SpringBootApplication`

public class ZipkinApplication {

 public static void main(String[] args) {

 SpringApplication.run(ZipkinApplication.class, args);

 }

}

❶ 这告诉 Zipkin 服务器监听传入的 span。

将 Zipkin UI (`io.zipkin:zipkin-ui`) 添加到 Zipkin Stream 服务器的类路径中以可视化请求。调出用户界面（也位于 `http://localhost:9411`，即 Stream 服务器所在的地方）。如果有的话，你可以找到所有最近的跟踪。如果没有，我们可以来创建。

随着服务器的启动和运行，我们可以启动几个客户端并发送一些请求。我们来看两个小服务，将它们命名为 `zipkin-client-a` 和 `zipkin-client-b`。这两个服务都有必需的绑定器 (`org.springframework.cloud:spring-cloud-starter-stream-rabbit`)，类路径上有 Spring Cloud Sleuth Stream 客户端 (`org.springframework.cloud:spring-cloud-sleuth-stream`)。

客户端 `zipkin-client-a` 被配置为在端口 8082 上运行。有一个属性 `message-service`，告诉客户端在哪里找到它的服务。不过，你也可以在这里轻松使用服务注册和发现。客户端使用在主类中定义的 `RestTemplate` bean 来请求下游服务。客户端（在本例中为 `RestTemplate`）是一个 Spring bean。如果它能够为流经它的所有请求配置一个 Sleuth 侦听拦截器（见示例 13-33），Spring Cloud Sleuth 配置需要知道在哪里可以找到这个 bean。

示例13-33 消息客户端

```
package demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import java.util.Map;

@RestController
class MessageClientRestController {

    @Autowired
    private RestTemplate restTemplate;

    @Value("${message-service}")
    private String host;

    @RequestMapping("/")
    Map<String, String> message() {

        //@formatter:off
        ParameterizedTypeReference<Map<String, String>> ptr =
            new ParameterizedTypeReference<Map<String, String>>() {};
        //@formatter:on

        return this.restTemplate.exchange(this.host, HttpMethod.GET, null, ptr)
            .getBody();
    }
}
```

服务 zipkin-client-b 被配置为在端口 8081 上运行。它从入站请求中获取所有跟踪头信息，并将其包含在回复中，并附带一条消息（见示例 13-34）。

示例13-34 消息服务

```
package demo;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletRequest;
import java.util.Collections;
```



```

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@RestController
class MessageServiceRestController {

    @RequestMapping("/")
    Map<String, String> message(HttpServletRequest httpServletRequest) {

        List<String> traceHeaders = Collections.list(httpServletRequest.getHeaderNames())
            .stream().filter(h -> h.toLowerCase().startsWith("x-"))
            .collect(Collectors.toList()); ❶

        Map<String, String> response = new HashMap<>();
        response.put("message", "Hi, @ " + System.currentTimeMillis());
        traceHeaders.forEach(h -> response.put(h, httpServletRequest.getHeader(h)));
        return response;
    }
}

```

- ❶ 从 zipkin-client-a 的外发请求中收集 Spring Cloud Sleuth 提供的所有 header (以 x- 开头), 并将其包含在生成的 JSON 响应中, 并附带一个唯一的消息。

向 <http://localhost:8082> 上发送几个请求, 会得到示例 13-35 所示的回复。

示例13-35 来自跟踪请求的示例回复

```

{
  "x-b3-parentspanid" : "9aa83c71878b6cd4",
  "x-b3-sampled" : "1",
  "message" : "Hi, 1493358280026",
  "x-b3-traceid" : "9aa83c71878b6cd4",
  "x-span-name" : "http:",
  "x-b3-spanid" : "668b8e088a35f1db"
}

```

现在可以在 <http://localhost:9411> 上查看 Zipkin 服务器中的请求。可以按最近、最长等排序来查看结果。如图 13-7 所示, 我们可以在 Zipkin 服务器中搜索跟踪结果。

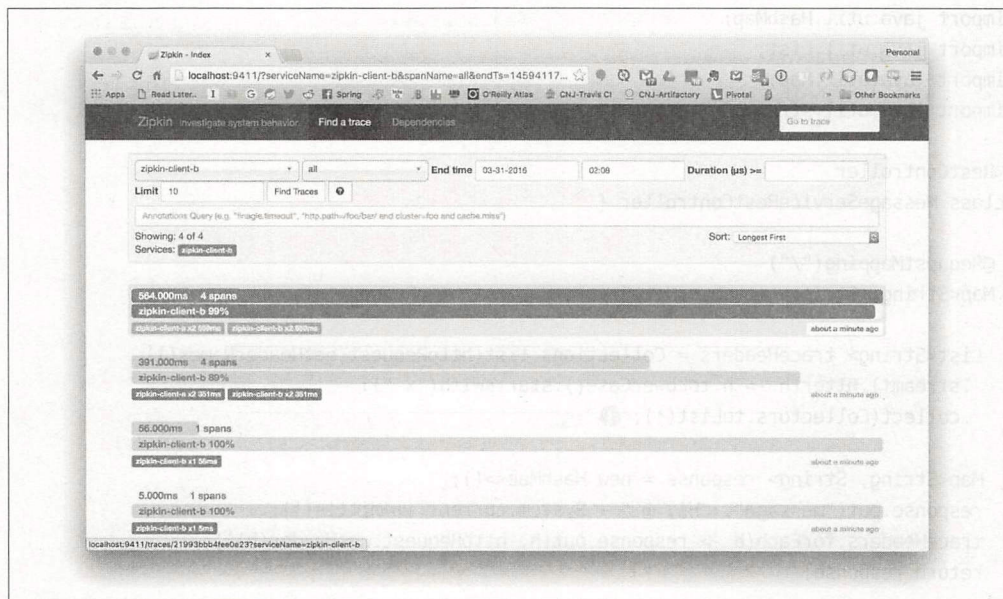


图13-7 在Zipkin主页上搜索跟踪的结果

可以检查跟踪的详细信息，如图 13-8 所示。

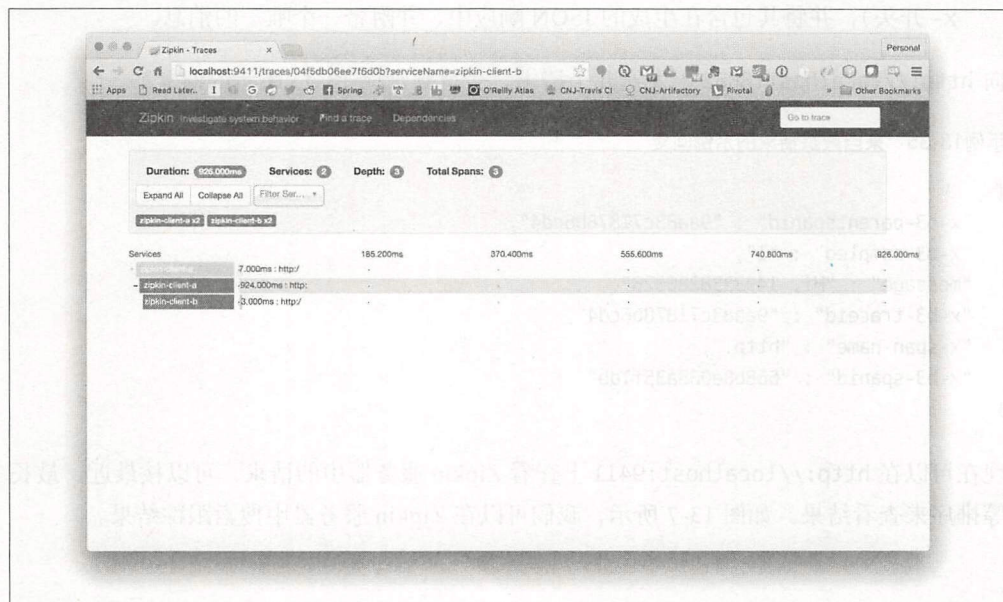


图13-8 详细信息页面显示单个跟踪的单个span

每个单独的 span 还带有与其相关的特定请求的信息 (tags)。可以通过单击单个 span 来查看详细信息，如图 13-9 所示。

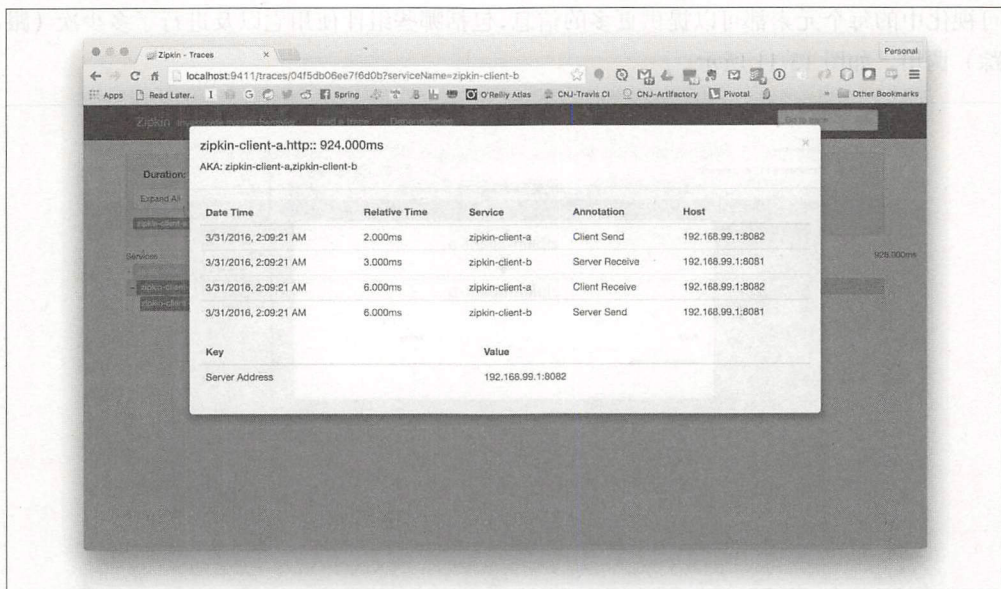


图13-9 详细信息面板显示给定span的相关信息和关联标记

Zipkin 还有一个强大的功能。它知道服务如何相互影响。它知道你的系统的拓扑结构。如果单击 Dependencies 选项卡，那么它会生成该拓扑的可视化表示，如图 13-10 所示。

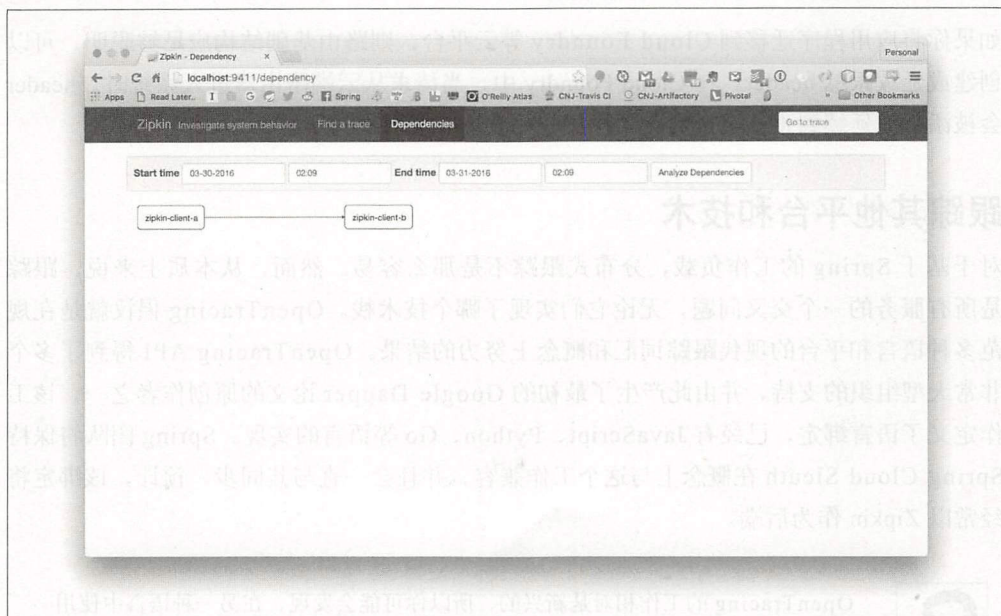


图13-10 服务拓扑结构的可视化

可视化中的每个元素都可以提供更多的信息,包括哪些组件使用它以及进行了多少次(跟踪)调用,如图 13-11 所示。

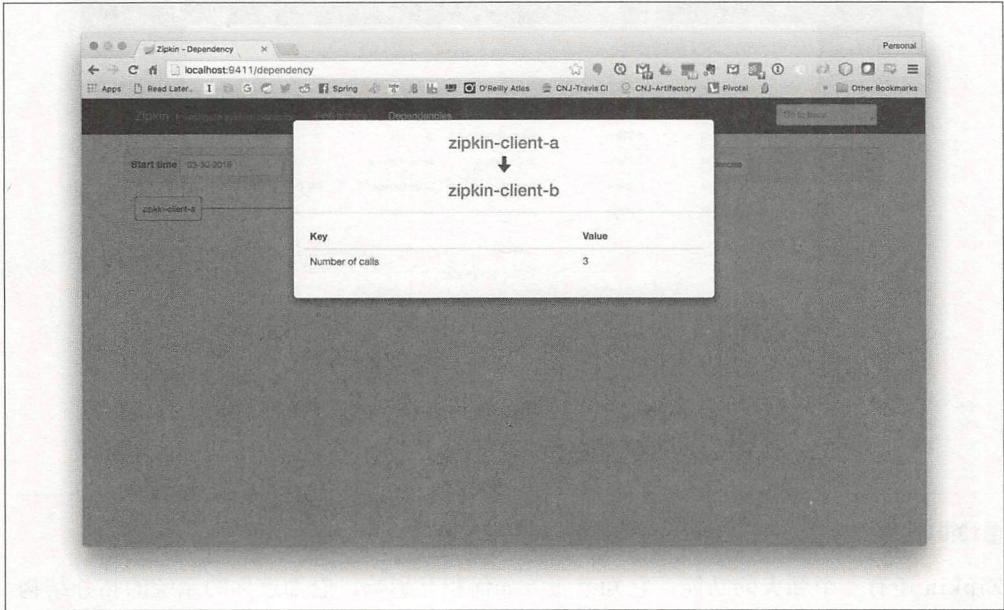


图13-11 依赖可视化显示中每个服务的详细信息

如果你将应用程序迁移到 Cloud Foundry 等云平台,则路由基础结构应足够聪明,可以创建或延续跟踪 header。在 Cloud Foundry 中,当请求从云端路由器进入系统时,header 会被添加或延续到正在运行的应用程序(如 Spring 应用程序)中。

跟踪其他平台和技术

对于基于 Spring 的工作负载,分布式跟踪不是那么容易。然而,从本质上来说,跟踪是所有服务的一个交叉问题,无论它们实现了哪个技术栈。OpenTracing 倡议就是在规范多种语言和平台的现代跟踪词汇和概念上努力的结果。OpenTracing API 得到了多个非常大型组织的支持,并由此产生了最初的 Google Dapper 论文的原创作者之一。该工作定义了语言绑定,已经有 JavaScript、Python、Go 等语言的实现。Spring 团队将保持 Spring Cloud Sleuth 在概念上与这个工作兼容,并且会一直与其同步。预计,该绑定将经常以 Zipkin 作为后端。



OpenTracing 的工作相对是新兴的,所以你可能会发现,在另一种语言中使用 OpenZipkin 客户端绑定更简单,而不是基于 OpenTracing 的实现。

仪表板

到目前为止，我们主要研究了如何为每个节点提供信息，以及如何定制这些信息。但是，我们还需要将它连接到更大的系统构成更全局的图景。举例来说，Actuator 发布给定节点的信息，但假设有某种基础设施来吸收这些信息并整合它，这类似于谷歌使用其 Borg 监控（“Borgmon”）方法管理服务的方式。Borgmon 是 Google 内部使用的集中管理和监控的解决方案，但它要求每个节点公开服务信息。感知 Borgmon 的服务通过 HTTP 端点发布信息，即使它们监视的服务本身不是 HTTP。除了 Actuator 提供的逐节点端点外，本章还将介绍几个关于如何集中和可视化系统本身的选项。

在本节中，我们将介绍几种方便的工具，以支持运维和业务人员都喜欢的仪表板体验。这些仪表板通常以我们所看到的工具为基础，以一目了然的方式呈现相关信息。这些工具依靠服务注册和发现来发现系统中的服务，然后显示关于它们的信息。有关使用 Netflix 的 Eureka 等服务注册表的详细信息，请参见第 7 章的内容。在本节中，仪表板依赖 Netflix Eureka 注册表，发现和监视系统中部署的服务。我们也可以使用 Hashicorp Consul 或者 Apache Zookeeper，或者其他可以使用 Spring Cloud DiscoveryClient 抽象实现的注册表。

使用 Hystrix 仪表板监控下游服务

我们无法在其他团队的代码中添加说明。我们不能坚持使用 Cloud Foundry 和 Spring Cloud 等最佳技术构建应用程序。通常我们不能让其他团队和其他组织做任何事情。我们所能做的就是保护自己免受潜在的下游代码失败的影响。为此，一种方法是使用断路器包装潜在的不稳定的服务间调用。

Spring Cloud 可以与 Netflix Hystrix 断路器轻松集成。我们在第 12 章中讨论了它。我们来看一个简单的例子，它在向 <http://google.com> 或 <http://yahoo.com> 发出调用时会随机插入失败处理（见示例 13-36）。

示例13-36 使用Hystrix断路器（假设你已经指定了@EnableCircuitBreaker）

```
package com.example;
```

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;
```

```
import java.net.URI;
import java.util.Random;
```

```
@RestController
```

```
class ShakyRestController {
```

```
    @Autowired
```

```
    private RestTemplate restTemplate;
```

❶

```
    public ResponseEntity<String> fallback() {
        return ResponseEntity.ok("ONONES");
    }
```

❷

```
    @HystrixCommand(fallbackMethod = "fallback")
    @RequestMapping(method = RequestMethod.GET, value = "/google")
    public ResponseEntity<String> google() {
        return this.proxy(URI.create("http://www.google.com/"));
    }
```

```
    @HystrixCommand(fallbackMethod = "fallback")
    @RequestMapping(method = RequestMethod.GET, value = "/yahoo")
    public ResponseEntity<String> yahoo() {
        return this.proxy(URI.create("http://www.yahoo.com/"));
    }
```

```
    private ResponseEntity<String> proxy(URI url) {
```

```
        if (new Random().nextInt(100) > 50) {
            throw new RuntimeException("tripping circuit breaker!");
        }
```

```
        ResponseEntity<String> responseEntity = this.restTemplate.getForEntity(url,
            String.class);
```

```
        return ResponseEntity.ok()
            .contentType(responseEntity.getHeaders().getContentType())
            .body(responseEntity.getBody());
    }
```

```
}
```

- ❶ 提供回退行为，当出现异常时被调用。在这种情况下，我们返回一个字符串。
- ❷ 使用断路器来装饰我们的各种 REST 调用，以便安全地处理可能失败的下游服务调用。

使用 Hystrix 断路器的每个节点还为包含断路器的每个节点发出服务器发送事件 (SSE) 心跳流。SSE 流可以从 `http://localhost:8000/hystrix.stream` 访问, 在默认配置下, 上面的代码侦听端口 8000。该流不断更新。它包含有关通过该线路的流量的信息, 包括请求的数量, 线路是否打开 (在这种情况下, 请求失败并被转移到 *fallback* 处理程序) 或关闭 (因此请求成功到达下游服务), 以及流量本身的统计。虽然我们无法监督其他人的服务, 但我们可以通过断路器监视请求的流量, 作为下游服务的一种虚拟监视器。如果断路器是断开的, 而且请求没有经过, 则可能表示下游业务已经中断。

我们可以监控断路器。断路器流并不是我们直接看的东西, 但是你可以把这个流发送给另一个组件 Hystrix Dashboard, 通过它可以看到通过特定节点的可视化请求流。

创建一个新的 Spring Boot 应用程序, 并在类路径中添加 `org.springframework.cloud:spring-cloud-starter-hystrix-dashboard`。将 `@EnableHystrixDashboard` 添加到 `@Configuration` 类, 然后启动应用程序。Hystrix 仪表板 UI 通过 `/hystrix.html` 访问。

不错! 但是, 这仍然只是一个节点。在规模上是站不住脚的, 你可能有多个相同服务的实例, 或者多个服务。单节点感知的 Hystrix 仪表板不会很有用, 你必须每次插入一个 `/hystrix.stream` 地址。我们可以使用 Spring Cloud Turbine 将来自所有节点的所有流复用为一个流。然后, 我们可以将生成的聚合流插入 Hystrix 仪表板。Spring Cloud Turbine 可以使用服务注册和发现 (通过 Spring Cloud DiscoveryClient 服务注册表抽象) 或者通过公开的 RabbitMQ 和 Apache Kafka 等消息传递代理 (通过 Spring Cloud Stream 消息抽象) 来聚合服务。

向新的 Spring Boot 应用程序添加 `org.springframework.boot:spring-boot-starter-web`、`org.springframework.cloud:spring-cloud-starter-stream-rabbit` 和 `org.springframework.cloud:spring-cloud-starter-turbine-stream` (见示例 13-37)。

示例13-37 使用Spring Cloud Turbine将来自多个断路器的服务器发送的事件心跳流跨多个节点聚合成一个流

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.turbine.stream.EnableTurbineStream;
```

```
1
@EnableTurbineStream
@SpringBootApplication
public class TurbineApplication {
```

```
public static void main(String[] args) {
    SpringApplication.run(TurbineApplication.class, args);
}
}
```

❶ 建立基于 Spring Cloud Stream 的涡轮聚合流

当 Spring Cloud Turbine 服务启动时，它将在 `http://localhost:8989/hystrix.stream` 处提供一个流，其中 8989 是默认端口。你可以通过指定 `turbine.stream.port` 来覆盖默认端口。本书的例子指定了 8010 端口。

所有有断路器的客户端都需要更新一下，以支持 Spring Cloud Turbine 的加入。添加 Spring Cloud Stream 绑定（我们使用 `spring-cloud-starter-stream-rabbit`），然后添加 `org.springframework.cloud:spring-cloud-netflix-hystrix-stream`。这个最后的依赖关系将断路器状态更新适配到通过特定的 Spring Cloud Stream 绑定程序选择发送的消息。作为其中的一部分，Spring Cloud Turbine 将需要关于本地节点和集群的信息。提供这些信息最简单的方法是使用服务注册和发现。在类路径中添加一个 `DiscoveryClient` 抽象实现（我们使用 `org.springframework.cloud:spring-cloud-starter-eureka`）。



这当然要求 Netflix 的 Eureka 服务注册表也在某处运行。有关更多详细信息，请参阅第 7 章中对服务注册和发现的讨论。当我们查看系统的观察方式时，我们将使用服务注册表，而不仅仅是单个节点。如果你从这里开始运行，你也可以让它保持运行。

重新启动客户端，然后重新访问断路器仪表板。从 Spring Cloud Turbine (`http://localhost:8010/hystrix.stream`，如果使用我们的代码) 插入 `hystrix.stream` 端点。图 13-12 显示了查看 Hystrix 流的结果。



像 Hystrix 仪表板这样的技术是非常重要的，但它们会增加运维成本。理想情况下，这个能力应该由平台来管理，并且是自动化的。如果你使用的是 Cloud Foundry，则服务目录中已有 Hystrix Dashboard 支持服务，因此可以使用 Spring Cloud Turbine。

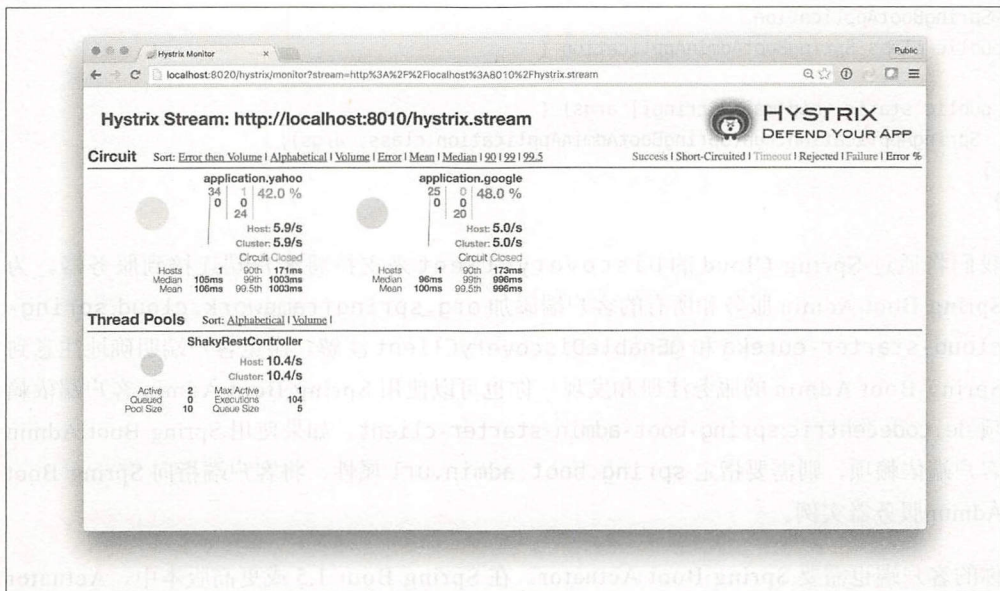


图13-12 Hystrix仪表板

Codecentric 的 Spring Boot Admin

Spring Boot Admin (<https://github.com/codecentric/spring-boot-admin>) 是 Codecentric 的一个项目。它提供了一个聚合的服务视图，并支持将其放入基于 Spring Boot 的服务的 Actuator 暴露的端点（日志、JMX 环境、请求日志等）上。

要使用它，需要启用服务注册表（我们正在使用预先描述的 Netflix Eureka 实例）。设置一个新的 Spring Boot 应用程序，并将以下依赖关系添加到类路径中：de.codecentric:spring-boot-admin-server 和 de.codecentric:spring-boot-admin-server-ui。当然版本本身会有所不同，所以请查看 Git 仓库和 Maven 仓库。在这个例子中，我们使用 1.5.0 版本（见示例 13-38）。

示例13-38 创建一个Spring Boot Admin的实例

```
package com.example;

import de.codecentric.boot.admin.config.EnableAdminServer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@EnableAdminServer
```

```
@SpringBootApplication
public class SpringBootAdminApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootAdminApplication.class, args);
    }
}
```

我们将通过 Spring Cloud 的 `DiscoveryClient` 来支持将客户端连接到服务器。为 Spring Boot Admin 服务和所有的客户端添加 `org.springframework.cloud:spring-cloud-starter-eureka` 和 `@EnableDiscoveryClient` 注解。避免客户端明确地注意到 Spring Boot Admin 的服务注册和发现。你也可以使用 Spring Boot Admin 客户端依赖项 `de.codecentric:spring-boot-admin-starter-client`。如果使用 Spring Boot Admin 客户端依赖项，则需要指定 `spring.boot.admin.url` 属性，将客户端指向 Spring Boot Admin 服务器实例。

你的客户端也需要 Spring Boot Actuator。在 Spring Boot 1.5 或更高版本中，Actuator 端点被锁定并需要认证。最简单的方法是禁用管理端点本身的身份验证（`management.security.enabled=false`）；否则 Spring Boot Admin 中的某些功能将不起作用。

启动你的客户端，然后访问 Spring Boot Admin（如图 13-13 所示）。

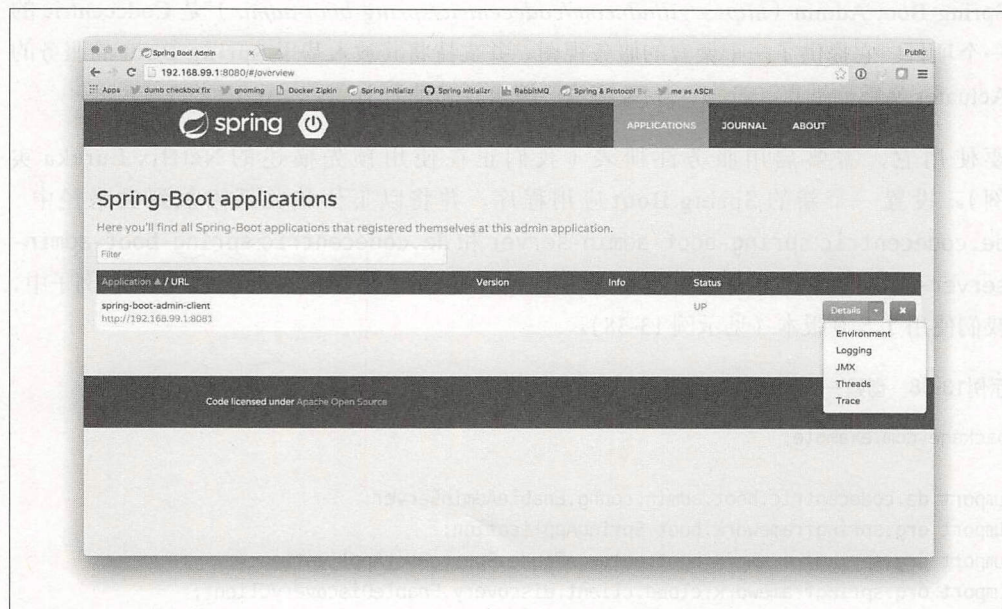


图13-13 列出已注册服务的Spring Boot Admin的界面

对于我们的例子，在 8080 端口上运行，如图 13-14 和图 13-15 所示。

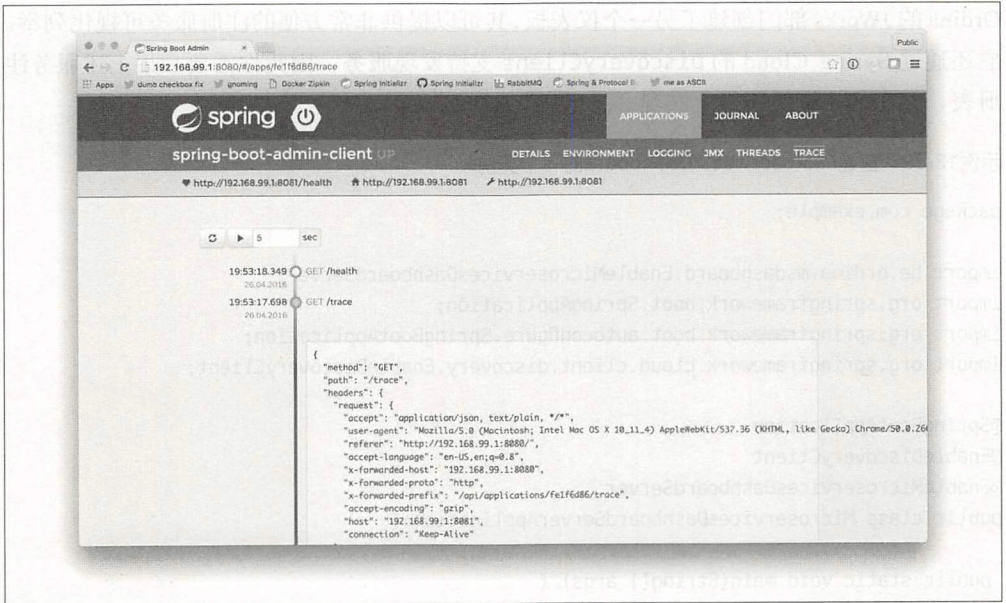


图13-14 Spring Boot Admin枚举请求 (/trace)

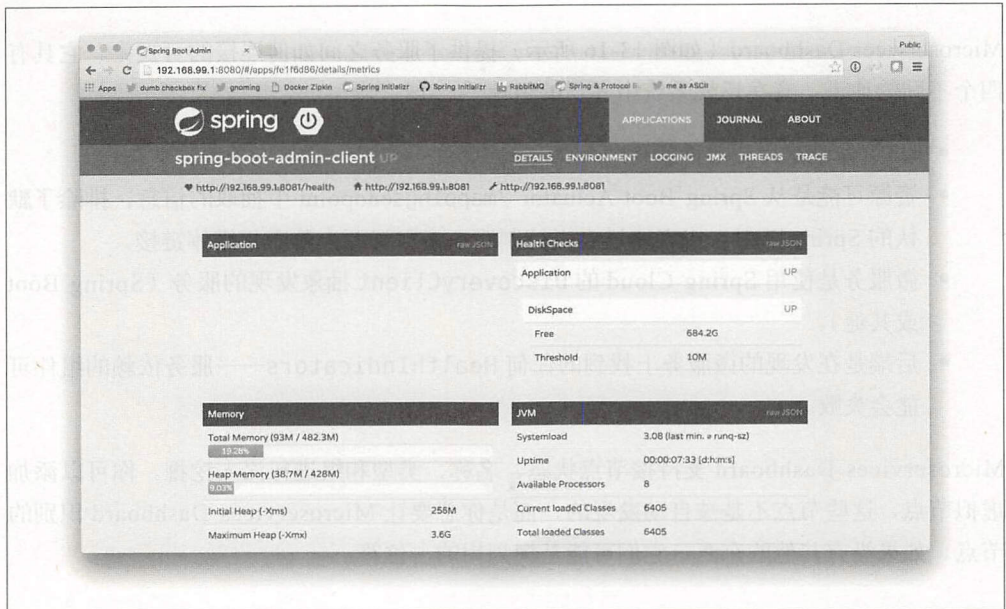


图13-15 Spring Boot Admin详细信息界面

Spring Boot Admin 提供了查看系统中服务的聚合情况并深入了解其状态的另一种方式。

Ordina Microservices 仪表板

Ordina 的 JWorks 部门创建了另一个仪表板,其可以提供非常方便的注册服务可视化列举。它还通过 Spring Cloud 的 `DiscoveryClient` 支持发现服务,因此你需要启用上述服务注册表,并在客户端类路径上实现(见示例 13-39)。

示例13-39 创建Microservices Dashboard的一个实例

```
package com.example;

import be.ordina.msdashboard.EnableMicroservicesDashboardServer;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
@EnableMicroservicesDashboardServer
public class MicroservicesDashboardServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(MicroservicesDashboardServerApplication.class, args);
    }
}
```

Microservices Dashboard (如图 13-16 所示) 提供了服务之间如何连接的可视化。它具有四个不同的通道,意在反映系统中各层的组件。

- UI 组件就是用户界面组件,例如 Angular。
- 资源可能是从 Spring Boot Actuator `/mappingsendpoint` 中抽取的信息,排除了默认的 Spring 映射,或者通过索引控制器在索引资源上暴露超媒体链接。
- 微服务是使用 Spring Cloud 的 `DiscoveryClient` 抽象发现的服务(Spring Boot 或其他)。
- 后端是在发现的微服务上找到的任何 `HealthIndicators`——服务依赖的组件可能会失败。

Microservices Dashboard 支持按节点状态、名称、类型和组进行深入挖掘。你可以添加虚拟节点,这些节点不是被自动发现的,而是你想要让 Microservices Dashboard 识别的节点。如果没有其他的东西,它们可能是规划用的占位符。

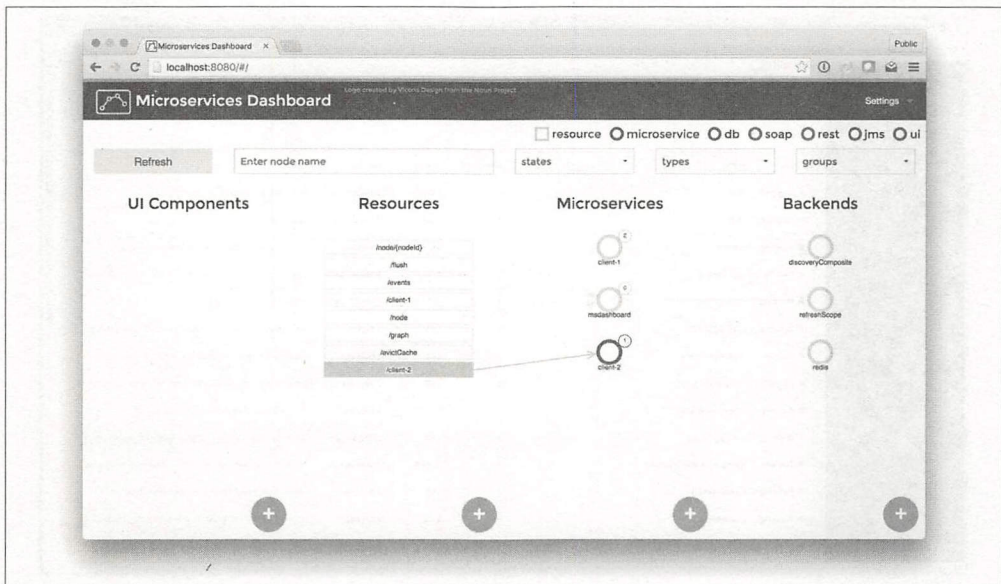


图13-16 Microservices Dashboard枚举发现的服务并提供这些服务中某些组件的信息

Pivotal Cloud Foundry 的 AppsManager

我们刚才看到的两个仪表板都依靠 Spring Boot 的 Actuator 来表示关于 Java 进程的信息。但是，最终，你将在平台上运行代码，并且在该平台中将会有其他可变组件，如运行应用程序本身的容器（反过来自己的健康状况将被监控）、支持服务等。如果你使用的是 Cloud Foundry 之类的东西，那么没有理由不能运行 .NET 应用程序或 Python 应用程序或其他任何你想要运行的应用程序，而且它们也将拥有自己的状态。可以说，最好的仪表板可以集中系统中所有应用程序的可视性。如果有更详细的诊断信息（例如 Spring Boot Actuator 的信息）可用，那么也应该清晰可见。这些 Pivotal Cloud Foundry AppsManager 都可以做到。我们已经在本章中介绍过了，所以这里我们只介绍 AppsManager 的概况。

我们可以查看属于特定组织和特定空间的给定用户的所有应用程序，如图 13-17 所示。



图13-17 查看属于特定组织和特定空间的给定用户的所有应用程序

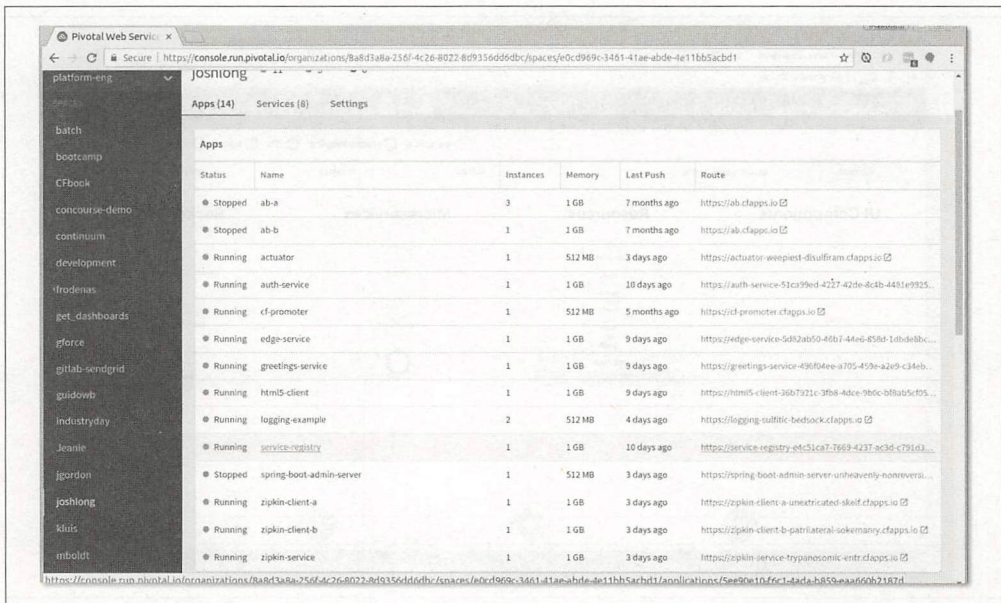


图13-17 特定Cloud Foundry空间中的所有应用程序，而该应用程序又是组织的一部分

我们也可以查看单个应用程序的详细信息，如图 13-18 所示。

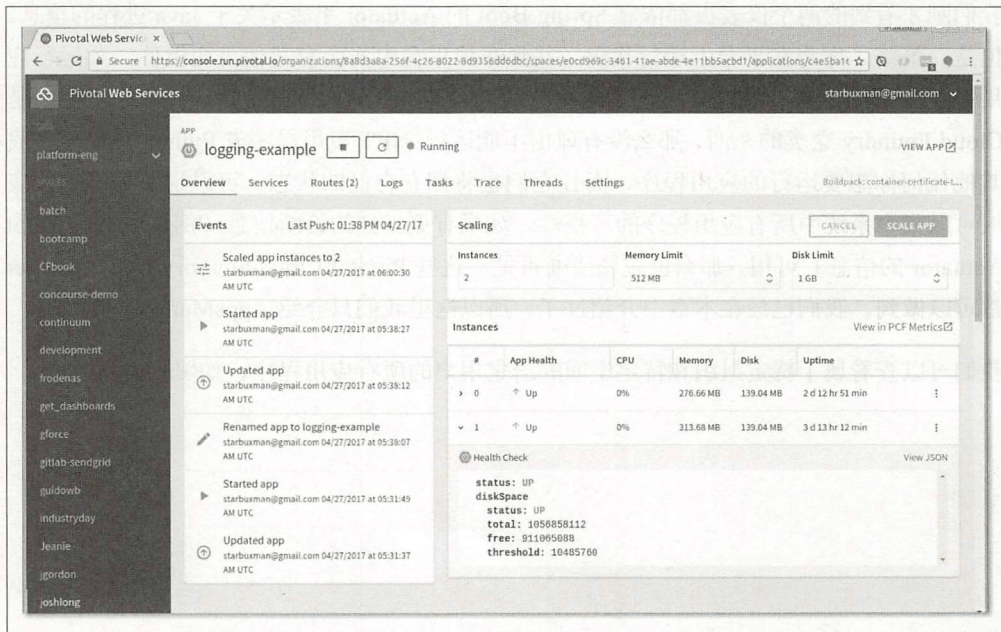


图13-18 特定Cloud Foundry应用程序的详细信息。Spring Boot Actuator的信息也是可见的

修复

前面我们一直着重于提供有关系统状态的信息，提高可视性。那么，我们如何处理这些知识呢？在一个静态的、云预备环境中，改进的可见性可以用来触发警报，然后（期望）寻呼机关闭或有人收到电子邮件。到发出警报时，就可能已经迟了，有的人已经对系统有了不好的体验。云计算（支持使用 API 进行操作）改变了这种状况：我们可以通过软件解决软件问题。例如，如果系统需要更多容量，则不需要提交 ITIL 单；只是创建一个 API 请求。我们可以支持自动修复。

分布式计算改变了我们在一个足够分散的系统中考虑应用程序实例可用性的方式，拥有一个 100% 可用的单一实例几乎是不可能的。相反，重点是建立一个服务能够在某种程度上得到恢复的系统。我们必须优化补救时间。如果修复的时间是零秒，那么我们可以构建有效的 100% 高度可用的系统，但这种方法的改变对我们如何构建系统产生了深远的影响。如果我们可以对平台进行编程，我们就能做到这一点。

该平台甚至可以自动为你做基本的修复。大多数云平台提供健康监测。如果你要求 Cloud Foundry 支持 10 个应用程序的实例，它将确保至少有 10 个实例。即使一个实例死亡，Cloud Foundry 也会重启。

包括 Pivotal Web Services 和 Pivotal Cloud Foundry 在内的大多数云平台都支持 *autoscaling*。Autoscaler 自动监控容器级别的信息，如 RAM 和 CPU，并在必要时通过启动新的应用程序实例来增加容量。在 Pivotal Web Services 上，你只需要创建一个 *app-autoscaler* 类型的服务实例（见示例 13-40），然后将其绑定到应用程序。你还需要在 Pivotal Web 服务控制台的管理面板中进行配置。

示例13-40 在Pivotal Web Services上创建一个autoscaler服务

```
cf marketplace
Getting services from marketplace in org platform-eng / space joshlong as ..
OK

service      plans      description
..
app-autoscaler  bronze, gold  Scales bound applications in response to load (beta)
..
```

还有进一步修复的余地。这里有一些令人振奋的例子，比如 Netflix 的 Winston (<http://bit.ly/2vncwz6>)、LinkedIn 的 Nerse (<http://bit.ly/2vnaMFX>)、Facebook 的 FBAR (<http://bit.ly/2vnp89p>) 和开源的 StackStorm (<https://stackstorm.com>)。这些工具可以很容易地定义由功能原子组成的管道，这些功能组合在一起解决问题。这些工具根据众所周知的输入事件进行工作，由监视代理或传感器或系统中部署的其他指示器触发。在传统架

构中，这些事件会触发警报，这很有用，但对于某类问题也可能触发自动修复流程。

我们鼓励你调研一些这类方法。它们大多数都根植于没有像 Cloud Foundry 这样的平台所提供保证的环境。对于我们的情况，我们不需要解决重启服务或负载均衡等问题。我们也不需要担心心跳检测等低级的事情。该平台将替我们管理。

尽管我们在可见性上仍然存在一些差距——我们只知道要寻找的东西，因为我们知道我们的架构的具体细节。在我们的架构中找到描述系统容量故事的组件并不难。这些组件公开了我们可以监控或反应的信息。Spring Integration 使处理来自不同事件源的事件变得更简单，并通过诸如 RabbitMQ 或 Apache Kafka 等消息传递技术将组件串起来。Spring Cloud Data Flow 建立在 Spring Integration 基础之上，提供了一个类似 Bash shell 的 DSL，使你可以组合任意的处理流，然后在分布式处理结构（如 YARN 或 Cloud Foundry）上编排它们。有关更多信息，请参阅第 10 章和第 12 章的内容。Spring Cloud Data Flow 是组装处理管道的理想工具箱，可响应来自我们自定义事件源的洞察。

你可能会看到很多东西。我们处于一个令人羡慕的位置——解决许多类别的问题都有 API。我们可以使用软件修复摇摇欲坠的软件。假设我们有一个消费者需要花费很多时间来回复消息。我们可以查看消息代理中队列的吞吐量，以决定是否应该添加更多消费者，来帮助更有效地消费队列，并保持在服务级别协议（SLA）中。

在本章的源代码中，有一些方便的 Spring Cloud Data Flow 源（*source*）（产生事件消息的组件）和汇聚（*sink*）（响应事件做某事的组件）。rabbit-queue-metrics 源监视给定的 RabbitMQ 队列并发布关于它的信息：队列深度（尚未处理的消息数量）、消费者数量（有多少客户端正在侦听另一端的消息）和队列名称本身。cloudfoundry-autoscaler 汇聚响应传入的消息（应该是一个数字），并为给定的应用程序添加或减少实例，直到该数目回落到规定的范围内。数字的范围和相关性是你自己定的。一旦注册了自定义的源和汇聚，就可以通过获取源的输出，从信息中提取队列大小，然后将其发送到 autoscaler 汇聚来连接这两件事情。

示例13-41 使用Spring Cloud Data Flow根据队列深度自动扩展应用程序

```
rabbit-queue-metrics
--rabbitmq.metrics.queueName=remediation-demo.remediation-demo
-group |
transform --expression=headers['queue-size'] |
cloudfoundry-autoscaler --applicationName=remediation-rabbitmq-consumer
--instance
CountMaximum=10 --thresholdMaximum=5
```

有很多输入变量可以用来理解如何支持和修复一个破坏的系统。如果你捕获和提取这些事件，并将它们连接到自动处理程序，那么你的系统将拥有针对某些简单类别问题的

件进行自动响应的基础。Spring Cloud Data Flow 是专门为这种特殊的事件监视和响应方式而建立的。在本章的源代码中，我们还包含了一个 Spring Cloud Data Flow *source* 组件来监视 Cloud Foundry 应用程序指标（如 RAM 和硬盘使用情况）。你可以根据应用程序的其他指标构建类似的修复流程。

总结

本章我们才开始了解各种可能性。如果你感到不知所措，好！可以从可视性的构架失败只会引发灾难这个方面来考虑，这个主题在云原生系统中很关键。这些问题通常被称为“第二天的问题”——直到生产之后，你才会意识到你需要的东西。如果你在“第一天”想到这些事情，那么在“第二天”就不那么痛苦了。Spring Boot、Spring Cloud 和 Cloud Foundry 是专门构建的，可以以最少的代价快速轻松地集成和支持这些需求。

你会注意到，尽管我们在本章中讨论的很多内容都是以开源客户端的形式出现，但是我们提到的一些支持服务是托管的 SaaS 产品，而且使用它们通常会花费很多成本。这是一个特点。如前所述，我们的目标是永远不要去构建我们不能销售的软件。最好让平台以及这些关注核心竞争力的第三方来满足这些非功能性需求。

考虑将所有这些以生产为中心的需求提取到一个单独的 Spring Boot 自动配置中，组织中的微服务依照它而构建。你甚至可以创建自己的 starter 依赖和元注释。这样，只需配置一次处理日志、度量和跟踪等的方式，然后只需将相应的自动配置添加到类路径中即可来使用。如果你使用 Cloud Foundry 这样的平台，构建支持应用程序的相关支持服务应该是轻而易举的事。总之，像 Spring Boot（支持十二要素的应用程序）和 Cloud Foundry（支持十二要素的运维）这样的应用程序框架可以帮助你安全快速地进行生产。

服务代理

后台服务 (backing service) 是应用程序在运行时通过网络使用的服务 (数据库、消息队列、电子邮件服务等)。Cloud Foundry 应用程序通过在 `VCAP_SERVICES` 环境变量中查找其定位器和凭证来消费后台服务。这种方法最大的特点就是简单。任何语言都可以读取环境变量,并解析环境变量中嵌入的 JSON 来提取诸如服务主机、端口和凭证之类的东西。应用程序的代码应该不用区分后台服务是由本地的还是由平台提供和管理的。这种解耦使得应用程序可以轻松地从在一个环境迁移到另一个环境:只需使用所在环境的值重新定义环境变量,然后重新启动应用程序即可。

在 Cloud Foundry 中,应用程序通过这种间接方式与所有服务进行通信。Cloud Foundry 管理一组后台服务和一组应用程序。运维人员必须指定哪些应用程序可以查看特定后台服务的连接信息。这被称为服务绑定 (service binding)。单个应用程序可能被绑定到多个后台服务,并且多个应用程序也可以被绑定到单个后台服务。在 Cloud Foundry 中提供绑定到应用程序的服务有两种方式:服务代理 (service broker) 和用户定义的服务 (user-defined service)。最终,它们都以 `VCAP_SERVICES` 环境变量中的条目的形式呈现。

如果运维人员希望应用程序与中间件或其他运行的平台之外的资源通信,运维人员可以创建用户定义的服务。这对于连接到在组织中运行的传统资源 (如 Ye Olde Sybase 实例或大型机) 或者其他 REST API (只是在某个平台上运行的应用程序) 而言尤其重要。

在使用通用资源并支持多租户的情况下,使用服务代理 (相对于用户定义的服务) 更有意义。服务代理本身是负责动态提供中间件实例并将连接信息提供给平台的应用程序,然后平台将其绑定到应用程序上。平台使用服务代理的方式来创建新的服务实例。服务代理还必须处理有问题的服务的生命周期,创建并在适当的时候销毁它。该平台通过统一的 REST API 与服务代理通信,所有服务代理都必须支持该 REST API。

创建后台服务

Cloud Foundry 服务市场（*services marketplace*）是暴露给平台用户的服务产品目录。运维人员可以以一致的方式在服务目录中创建任何服务。应用程序及其服务由平台统一调配。

使用 `cf create-service`（或者 `cf cs`）命令来创建一个新的服务实例（见示例 14-1）。

示例14-1 使用CF CLI创建一个新的服务

```
NAME:
    create-service - Create a service instance

USAGE:
    cf create-service SERVICE PLAN SERVICE_INSTANCE [-c PARAMETERS_AS_JSON] [-t TAGS]
```

可以看到，在示例 14-1 中，在创建一个新的服务实例时需要提供参数。第一个参数是市场目录中 *service* 的名称。`cf marketplace` 命令检索具有给定用户账户的可用套餐的服务列表。

在 Pivotal 托管的 Cloud Foundry 实例（称为 *Pivotal Web Services*）上运行此命令，你会发现市场中有一个 *rediscloud* 服务，并提供了多种套餐。市场中的 *rediscloud* 服务允许我们创建一个服务实例，它将提供一个 Redis 数据库部署，我们可以将它绑定到自己的应用程序上。使用命令 `cf marketplace -s rediscloud` 可以深入了解不同的 Redis 服务套餐（见示例 14-2）。

示例14-2 Redis部署的服务套餐列表

```
$ cf marketplace -s rediscloud

Getting service plan information for service rediscloud for user...
OK

service plan  description      free or paid
100mb         Basic             paid
250mb         Mini              paid
500mb         Small             paid
1gb           Standard          paid
2-5gb         Medium            paid
5gb           Large             paid
10gb          X-Large           paid
50gb          XX-Large          paid
30mb          Lifetime Free Tier free
```

可以看到，有不同的付费层级的资源和一个免费提供的 30mb 套餐。我们来创建一个使用 30mb 的免费套餐的 Redis 服务实例（如示例 14-3 所示）。

示例14-3 创建一个新的Redis服务实例

```
cf create-service rediscloud 30mb my-redis
```

现在我们已经创建了一个新的服务实例（my-redis），我们可以将它绑定到一个现有的应用程序上。假设我们有一个名为 user-api 的应用程序，它包含一个 Spring Boot 应用程序，使用 Redis 来缓存存储在 MySQL 数据库中的用户。将其绑定到我们的 Spring Boot 应用程序后，我们可以自动连接到 Redis 服务实例。绑定到服务实例的命令是 cf bind-service（如示例 14-4 所示）。

示例14-4 将服务实例绑定到应用程序

NAME:

bind-service - Bind a service instance to an app

USAGE:

```
cf bind-service APP_NAME SERVICE_INSTANCE [-c PARAMETERS_AS_JSON]
```

我们知道，我们要绑定的应用程序名是 user-api，服务实例名是 my-redis，现在可以将我们的用户应用程序绑定到新创建的 Redis 数据库实例（如示例 14-5 所示）。

示例14-5 将my-redis服务实例绑定到user-api应用程序

```
cf bind-service user-api my-redis
```

将应用程序绑定到 Cloud Foundry 中的服务实例时，服务代理将提供一组向应用程序描述服务实例的环境变量。通过使用 cf env user-api 命令（见示例 14-6）查看 user-api 环境变量，我们可以看到 my-redis 绑定的结果。

示例14-6 显示user-api应用程序的环境变量

```
$ cf env user-api
```

Getting env variables for app user-api in org user-org...

OK

System-Provided:

```
{
  "VCAP_SERVICES": { ❶
    "rediscloud": [
      {
        "credentials": { ❷
          "hostname": "pub-redis-9809.us-east-1-3.7.ec2.redislabs.com",
          "password": "not-a-real-password",
          "port": "17709"
        },
        "label": "rediscloud",
        "name": "my-redis", ❸
        "plan": "30mb", ❹
      }
    ]
  }
```



```

"tags": [
  "Data Stores",
  "Data Store",
  "Caching",
  "Messaging and Queuing",
  "key-value",
  "caching",
  "redis"
]
}
]
}
}
}

```

- ❶ VCAP_SERVICES 描述应用的所有服务实例绑定。
- ❷ 提供的绑定服务实例的凭证。
- ❸ 我们从 rediscloud 创建的 my-redis 服务实例的名称。
- ❹ 我们在创建 my-redis 免费层实例时选择的计划。

现在，我们已经使用 rediscloud 服务代理调配了一个新的服务实例，并将其绑定到了我们的应用程序上，这与十二要素宣言 (<https://12factor.net/backing-services>) 中定义的后台服务的思想一致。这一节我们探讨了如何使用 Cloud Foundry 市场中的服务目录创建服务实例和绑定。还了解了服务代理是如何使用系统提供的 VCAP_SERVICES 环境变量将连接细节注入应用程序部署中的。

平台视图

现在我们来探讨 Cloud Foundry 平台背后的机制，之后我们就可以使用 Spring Boot 开发自己的服务代理。Cloud Foundry 由许多通过 HTTP 进行通信的 Web 服务模块组成，类似于本书中讨论过的一些分布式系统架构。这些服务组合作为一个统一的 REST API 对外公开——*Cloud Controller API*。Cloud Controller API 是客户端触发基础云供应商进行虚拟基础架构编排的主要接口。这是描述 Cloud Foundry 平台功能的主要协议。

Cloud Controller 还维护与 Cloud Foundry 交互时使用的许多域资源的数据库，例如组织、空间、用户角色、服务定义、服务实例、应用程序等。由于 Cloud Controller 是服务定义和服务实例的所有者，因此 Cloud Foundry 的这个模块允许外部服务代理的生命周期管理，采取数据库和其他中间件服务的形式。

服务代理 API 是一个 REST API，其描述了 Cloud Controller 对第三方服务代理应用程序的期望。通常，服务代理不属于 Cloud Foundry 发行版的一部分。只有在配置了 Cloud

Foundry 版本后才能添加代理。它们会作为支持服务附加在 Cloud Controller API 上。

服务代理描述了它可以提供的服务的目录。这些可以在 Cloud Foundry 市场中找到，你可以使用 `cf marketplace` 查看。目录描述一个或多个服务及其套餐。服务的套餐涉及不同的服务水平，通常与不同的成本相关联。运维人员可以通过指定服务的名称和相关的套餐来创建新的服务实例。你可以使用 `cf create-service` 来做到这一点。创建的服务可能表示已启动的虚拟机或已初始化的资源。运维人员可以（可选地）使用 `cf bind-service` 将服务实例绑定到他们的应用程序。在这里，服务代理必须提供应用程序可以用来连接到配置服务的凭证。服务代理还必须处理删除服务实例和服务实例绑定的事情。

表 14-1 显示了服务代理期望的 REST 端点。

表14-1

说明	路由	方法
读取目录中描述的服务代理	<code>/v2/ catalog</code>	GET
删除服务代理和应用程序之间的绑定	<code>/v2/service_instances/{instanceId}/service_bindings/{bindingId}</code>	DELETE
在应用程序和服务之间创建一个新的服务实例绑定	<code>/v2/service_instances/{instanceId}/service_bindings/{bindingId}</code>	PUT
检索服务代理执行的最后一个操作的状态	<code>/v2/service_instances/{instanceId}/last_operation</code>	GET
检索服务实例	<code>/v2/service_instances/{instanceId}</code>	GET
创建一个新的服务实例	<code>/v2/service_instances/{instanceId}</code>	PUT
删除服务实例	<code>/v2/service_instances/{instanceId}</code>	DELETE
更新服务实例	<code>/v2/service_instances/{instanceId}</code>	PATCH

使用 Spring Cloud Cloud Foundry Service Broker 实现服务代理

在本节中，我们将利用通过本书学到的内容，使用 Spring Boot 创建一个服务代理。创建服务代理是扩展和自定义 Cloud Foundry 平台很重要的部分。有各种不同语言和平台的开源示例，演示了如何创建服务代理。我们将使用 Spring Cloud Cloud Foundry Service Broker 项目 (<http://cloud.spring.io/spring-cloud-cloudfoundry-service-broker/>) 来简化构建服务代理的非功能性需求工作：管理服务实例和服务实例绑定的生命周期。

简单的 Amazon S3 服务代理

Amazon Web Services (AWS) 提供了一个丰富的服务生态系统，即使我们不在该平台上

运行，我们的应用程序也可以利用这些服务。Amazon S3 对于保存数据非常有用。云原生应用程序的核心原则是应用程序实例是无状态的。我们不能保证正在运行我们的应用程序的虚拟机实例永远都不变，我们不能确定应用程序实例的文件系统就在那里。S3 提供了一种将数据存储作为支持服务附加到应用程序的方法。Amazon S3 不是你的操作系统可以理解的文件系统，因此它不能直接替代 JDK 的 `java.io.File`。它有自己的 API，实现了相同的目标，因此应用程序可以轻松地与 Amazon S3 进行交互，这对将现有应用程序迁移到云中非常有用。服务代理是集成传统（但有时是必需的）服务（如 FTP、文件系统和电子邮件）的好方法，实际上，在你的组织中还没有平台支持的多租户服务。

实现服务代理包含两个任务：提供服务代表的资源；以及实现服务代理 REST API。如果我们满足三个关键组件，Spring Cloud Cloud Foundry Service Broker 项目将提供 REST API。

- Catalog bean 的一个实例，它向 Cloud Controller 描述此服务代理提供的可用服务。
- 一个 `ServiceInstanceService` 的实现来管理创建和销毁服务实例。
- 一个 `ServiceInstanceBindingService` 的实现来管理创建和销毁服务实例绑定。

服务目录

实现服务代理的第一步是创建一个服务目录（*service catalog*），以描述你的代理将执行的操作。该目录还将提供一系列服务计划，来描述支持服务将如何创建和运行的各个方面。我们以通过服务代理提供作为服务的数据库为例。我们的 Amazon S3 代理将只提供一个单一的计划，这是一个在逻辑上（可能不是经济上的！）无限制的文件存储（见示例 14-7）的 S3 存储桶。

示例14-7 代理服务目录的接口协议

```
package cnj;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.cloud.servicebroker.model.Catalog;
import org.springframework.cloud.servicebroker.model.Plan;
import org.springframework.cloud.servicebroker.model.ServiceDefinition;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import java.util.Collections;
import java.util.List;
import java.util.Map;
```

```
@Configuration
```

```

class CatalogConfiguration {❶

    private final String spacePrefix;

    @Autowired
    CatalogConfiguration(
        @Value("${vcap.application.space_id:${USER:}}") String spaceId) {
        this.spacePrefix = spaceId + '-';
    }

    @Bean
    Catalog catalog() {
        Plan basic = buildBasicPlan();❷
        ServiceDefinition definition = buildS3ServiceDefinition(basic);❸
        return new Catalog(Collections.singletonList(definition));
    }

    private ServiceDefinition buildS3ServiceDefinition(Plan basic) {
        String description = "Provides AWS S3 access";
        String id = this.spacePrefix + "1489291412183";
        String name = "s3-service-broker";
        boolean bindable = true;
        List<Plan> plans = Collections.singletonList(basic);
        return new ServiceDefinition(id, name, description, bindable, plans);
    }

    private Plan buildBasicPlan() {
        String planName = "basic";
        String planDescription = "Amazon S3 bucket with unlimited storage";
        boolean free = true;
        boolean bindable = true;
        String planId = this.spacePrefix + "249722552510577";
        Map<String, Object> metadata = Collections.singletonMap("costs",
            Collections.singletonMap("free", true));
        return new Plan(planId, planName, planDescription, metadata, free, bindable);
    }
}

```

- ❶ 服务代理服务 and 计划必须具有唯一的 ID，该 ID 对于你要部署的 Cloud Foundry 实例的整体而言是全局唯一的。所以在这里，我们执行一些操作，把 Cloud Foundry 空间（或者至少是用户名）作为静态 ID 的前缀。这些值是随机的，并且是提前生成的。因此，假设你在每个空间只部署一次服务代理（这里仅仅是假设），服务目录应该是唯一的。如果要更新服务代理，则有必要使用稳定的服务 ID；否则，只需使用 UUID。

- ② 创建一个简单的套餐计划实例（这是免费的）。
- ③ 然后将计划注册为我们的一个服务定义的一部分。一个服务代理肯定可以托管多个服务定义（例如，Amazon Web Services 目录中的每个服务都有一个定义）。

为了简洁起见，目录的配置被硬编码到应用程序中，尽管可以在外部配置中维护或与 Spring Cloud Config Server 集成。

管理服务实例

Spring Cloud Cloud Foundry Service Broker 希望实现接口 `ServiceInstanceService`（见示例 14-8）。

示例14-8 `ServiceInstanceService`的定义

```
public interface ServiceInstanceService {
    CreateServiceInstanceResponse createServiceInstance(
        CreateServiceInstanceRequest r);
    UpdateServiceInstanceResponse updateServiceInstance(
        UpdateServiceInstanceRequest r);
    DeleteServiceInstanceResponse deleteServiceInstance(
        DeleteServiceInstanceRequest r);
    GetLastServiceOperationResponse getLastOperation(
        GetLastServiceOperationRequest r);
}
```

除了 `GetLastServiceOperationRequest` 外，其他方法应该是非常简单的。此方法用于返回异步操作的状态。在默认情况下，服务代理是同步的：当工作完成时，返回对 REST 端点的调用。但情况并非总是如此：假设一个服务代理用来提供 Hadoop 集群或 Spark 集群，或者尝试在删除调用中拆除这样的集群。这项工作需要的可能比服务代理所允许的任何合理超时时间长得多。在我们的例子中，服务调用总是成功返回的，对 Amazon Web Services 的调用并不费时。

我们的 `ServiceInstanceService` 实现将响应这些请求，并将相关信息保存在 SQL 数据库（特别是 MySQL）中，以便我们可以将请求与实际的服务实例关联到由此服务代理管理的资源上（见示例 14-9）。

示例14-9 `ServiceInstance`的定义

```
package cnj;

import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
```

```
import lombok.ToString;
import org.springframework.cloud.servicebroker.model.CreateServiceInstanceRequest;
import org.springframework.cloud.servicebroker.model.DeleteServiceInstanceRequest;
import org.springframework.cloud.servicebroker.model.UpdateServiceInstanceRequest;
```

```
import javax.persistence.Entity;
import javax.persistence.Id;
```

```
@Entity
```

①

```
@Data
```

②

```
@NoArgsConstructor
```

```
@ToString
```

```
@EqualsAndHashCode
```

```
class ServiceInstance {
```

```
    @Id
```

```
    private String id;
```

```
    private String serviceDefinitionId;
```

```
    private String planId;
```

```
    private String organizationGuid;
```

```
    private String spaceGuid;
```

```
    private String dashboardUrl;
```

```
    private String username, accessKeyId, secretAccessKey;
```

③

```
    public ServiceInstance(CreateServiceInstanceRequest request) {
        this.serviceDefinitionId = request.getServiceDefinitionId();
        this.planId = request.getPlanId();
        this.organizationGuid = request.getOrganizationGuid();
        this.spaceGuid = request.getSpaceGuid();
        this.id = request.getServiceInstanceId();
    }
```

```
    public ServiceInstance(DeleteServiceInstanceRequest request) {
        this.id = request.getServiceInstanceId();
        this.planId = request.getPlanId();
        this.serviceDefinitionId = request.getServiceDefinitionId();
    }
```



```

public ServiceInstance(UpdateServiceInstanceRequest request) {
    this.id = request.getServiceInstanceId();
    this.planId = request.getPlanId();
}
}

```

- ❶ ServiceInstance 是一个 JPA 实体。
- ❷ 使用 Lombok 项目的 @Data 构造函数来消除烦琐的 getter 和 setter 代码生成，无参构造函数 toString 方法和 equals /hashCode 对等。
- ❸ 支持创建、删除和更新请求的各种复制构造函数。

我们需要把这些细节保存在一个 SQL 数据库中（在这个项目中是 MySQL）。我们使用 Spring Data JPA 存储库来实现（见示例 14-10）。

示例14-10 ServiceInstanceRepository的定义

```

package cnj;

import org.springframework.data.jpa.repository.JpaRepository;

interface ServiceInstanceRepository extends
    JpaRepository<ServiceInstance, String> {
}

```

在创建新服务实例时，我们需要集成使用 Java SDK 的 AWS，以管理 IAM 用户和 S3 存储桶的创建。这是代理最重要的部分。该逻辑的大部分都位于名为 S3Service 的代码库中的定制服务实现中。我们不会在这里显示它的代码，请参阅本书的源代码，因为这是使用 AWS Java SDK 构建的比较乏味的逻辑。唯一值得理解的是，AWS Java SDK 需要授权，配置如示例 14-11 所示。

示例14-11 包含已验证的AmazonS3客户端的bean定义的配置类

```

package cnj.s3;

import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.identitymanagement.AmazonIdentityManagementClient;
import com.amazonaws.services.s3.AmazonS3Client;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
class S3Configuration {

```

```

@Bean
AmazonS3Client amazonS3Client(BasicAWSCredentials credentials) {
    return new AmazonS3Client(credentials);
}

@Bean
AmazonIdentityManagementClient identityManagementClient(
    BasicAWSCredentials awsCredentials) {
    return new AmazonIdentityManagementClient(awsCredentials);
}

❶
@Bean
BasicAWSCredentials awsCredentials(
    @Value("${aws.access-key-id}") String awsAccessKeyId,
    @Value("${aws.secret-access-key}") String awsSecretAccessKey) {
    return new BasicAWSCredentials(awsAccessKeyId, awsSecretAccessKey);
}

@Bean
S3Service s3Service(AmazonIdentityManagementClient awsId, AmazonS3Client s3) {
    return new S3Service(awsId, s3);
}
}

```

- ❶ 为了实现此目的，需要配置 Amazon Web Services 访问令牌和 Amazon Web Services 访问权限。在我们的持续集成环境中，这是作为一个环境变量提供的。

通过使用我们的 S3Service, ServiceInstanceService 实现变得快速了。在 S3 的情况下，真的没有什么需要提供的——AWS 已经有了 S3 可以直接使用。所以创建服务实例的大部分操作与创建一个持有其身份的 S3 用户有关。稍后，当我们将应用程序绑定到已配置的 S3 实例时，我们将调取持久化的信息，并将其提供给 Cloud Controller 以在运行中的应用程序的环境中公开。这意味着所有绑定的应用程序都将具有相同的凭证。这可能是，也可能不是你想要的，但是如果你信任开发应用程序的各方，这是一个明智的行为（见示例 14-12）。

示例14-12 DefaultServiceInstanceService的定义

```

package cnj;

import cnj.s3.S3Service;
import cnj.s3.S3User;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;

```



```

import org.springframework.cloud.servicebroker.exception.ServiceBrokerException;
import org.springframework.cloud.servicebroker.model.*;
import org.springframework.cloud.servicebroker.service.ServiceInstanceService;
import org.springframework.stereotype.Service;

@Service
class DefaultServiceInstanceService implements ServiceInstanceService {

    private final S3Service s3Service;

    private final ServiceInstanceRepository instanceRepository;

    private Log log = LoggerFactory.getLog(getClass());

    @Autowired
    DefaultServiceInstanceService(S3Service s3Service,
        ServiceInstanceRepository instanceRepository) {
        this.s3Service = s3Service;
        this.instanceRepository = instanceRepository;
    }

    @Override
    public CreateServiceInstanceResponse createServiceInstance(
        CreateServiceInstanceRequest request) {
        if (!this.exists(request.getServiceInstanceId())) {
            ServiceInstance si = new ServiceInstance(request);
            S3User user = s3Service.createS3UserAndBucket(si.getId()); ❶
            si.setSecretAccessKey(user.getAccessKeySecret());
            si.setUsername(user.getUsername());
            si.setAccessKeyId(user.getAccessKeyId());
            this.instanceRepository.save(si);
        }
        else {
            this.error("could not create serviceInstance "
                + request.getServiceInstanceId());
        }
        return new CreateServiceInstanceResponse();
    }

    @Override
    public DeleteServiceInstanceResponse deleteServiceInstance(
        DeleteServiceInstanceRequest request) {
        String sid = request.getServiceInstanceId();
        if (this.exists(sid)) {
            ServiceInstance si = this.instanceRepository.findOne(sid);
            ❷
            boolean deleteSucceeded = this.s3Service.deleteBucket(si.getId()),

```

```

        si.getAccessKeyId(), si.getUsername());
    if (!deleteSucceeded) {
        log.error("could not delete the S3 bucket for service instance " + sid);
    }
    this.instanceRepository.delete(si.getId());
}
else {
    this.error("could not delete the S3 service instance " + sid);
}
return new DeleteServiceInstanceResponse();
}

```

③

```

@Override
public UpdateServiceInstanceResponse updateServiceInstance(
    UpdateServiceInstanceRequest request) {
    String sid = request.getServiceInstanceId();
    if (this.exists(sid)) {
        ServiceInstance instance = this.instanceRepository.findOne(sid);
        this.instanceRepository.delete(instance);
        this.instanceRepository.save(new ServiceInstance(request));
    }
    else {
        this.error("could not update serviceInstance " + sid);
    }
    return new UpdateServiceInstanceResponse();
}

@Override
public GetLastServiceOperationResponse getLastOperation(
    GetLastServiceOperationRequest request) {
    return new GetLastServiceOperationResponse()
        .withOperationState(OperationState.SUCCEEDED);
}

private void error(String msg) {
    throw new ServiceBrokerException(msg);
}

private boolean exists(String serviceInstanceId) {
    return instanceRepository.exists(serviceInstanceId);
}
}

```

- ① 这种方法大部分是执行簿记工作的，除了这一行，通过 S3Service 与 Amazon Web Services API 交互来创建一个 S3 存储桶和一个可以访问该存储桶的 S3User。

- ② 删除操作将撤消创建操作，删除用户和存储桶。
- ③ 更新操作对实际调配的 S3 存储桶或用户没有任何作用，只是更新关于服务实例本身的持久化元数据。

后台服务的作用是将它的所有必要信息都作为环境变量注入应用程序的容器中。这个过程就是我们在 Cloud Foundry 中所称的绑定到服务实例的过程。绑定到服务实例的结果是关于如何使用被注入到应用程序的容器中的服务的连接信息。当应用程序在容器中暂存并启动后将能够找到服务实例的连接信息的环境变量。然后，应用程序将使用这些信息来定位并连接到正在运行的服务实例。下面我们来看看如何将我们的服务实例绑定到后台应用程序。

服务绑定

确定通过平台提供的功能作为服务或在每个应用程序中独立地实现功能它们之间的界限通常会令人头疼。确定何时通过平台将你的应用程序中的功能委托给后台服务有一个很好的经验法则。

如果你需要在所有应用程序中实现相同的功能，则应通过该平台将该功能作为服务提供。

云原生应用程序开发的目标是减少在与应用程序业务逻辑无关的地方花费时间。该平台不仅仅是在云中运行应用程序的运维工具，它更是一台可以将应用程序转换为传统代码的机器，并将其作为可替换的服务提供。例如，架构中的许多应用程序可能需要存储文件的能力。通过为应用程序提供可以与 Amazon S3 存储桶连接的服务代理，每个应用程序就不需要花费时间来实现自定义存储代码。应用程序开发人员需要做的唯一事情就是将应用程序绑定到服务实例，并开始存储和检索文件。

在 Spring Cloud Cloud Foundry Service Broker 中，这个功能由 `ServiceInstanceBindingService` 实现提供（见示例 14-13）。

示例14-13 `ServiceInstanceBindingService`的定义

```
public interface ServiceInstanceBindingService {  
  
    CreateServiceInstanceBindingResponse createServiceInstanceBinding(  
        CreateServiceInstanceBindingRequest r);  
  
    void deleteServiceInstanceBinding(DeleteServiceInstanceBindingRequest r);  
}
```

和 `ServiceInstanceBindingService` 一样，我们需要记录关于应用程序绑定的信息。我

们使用另一个 JPA 实体 `ServiceInstanceBinding` 来执行此操作（见示例 14-14）。

示例14-14 `ServiceInstanceBinding`的定义

```
package cnj;

import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
@Data
@NoArgsConstructor
public class ServiceInstanceBinding {

    @Id
    private String id;

    private String serviceInstanceId, syslogDrainUrl, appGuid;

    public ServiceInstanceBinding(String id, String serviceInstanceId,
        String syslogDrainUrl, String appGuid) {
        this.id = id;
        this.serviceInstanceId = serviceInstanceId;
        this.syslogDrainUrl = syslogDrainUrl;
        this.appGuid = appGuid;
    }
}
```

我们将继续使用另一个 Spring Data JPA 存储库 `ServiceInstanceBindingRepository`（见示例 14-15）来保存 `ServiceInstanceBinding`。

示例14-15 定义用于管理服务实例实体的 `ServiceInstanceBindingRepository`

```
package cnj;

import org.springframework.data.jpa.repository.JpaRepository;

public interface ServiceInstanceBindingRepository extends
    JpaRepository<ServiceInstanceBinding, String> {

}
```

在我们的 `ServiceInstanceBindingService` 实现中，我们使用这两个存储库来完成大部分的工作。`DefaultServiceInstanceBindingService` 只需要从数据库中调用持久化的

S3 凭证并把它们带到客户端即可（见示例 14-16）。

示例14-16 DefaultServiceInstanceBindingService的定义

```
package cnj;

import org.springframework.beans.factory.annotation.Autowired;

//@formatter:off
import org.springframework.cloud.servicebroker.exception.
    ServiceInstanceBindingDoesNotExistException;
import org.springframework.cloud.servicebroker.exception.
    ServiceInstanceBindingExistsException;
import org.springframework.cloud.servicebroker.model
    .CreateServiceInstanceAppBindingResponse;
import org.springframework.cloud.servicebroker.model
    .CreateServiceInstanceBindingRequest;
import org.springframework.cloud.servicebroker.model
    .CreateServiceInstanceBindingResponse;
import org.springframework.cloud.servicebroker.model
    .DeleteServiceInstanceBindingRequest;
import org.springframework.cloud.servicebroker.service
    .ServiceInstanceBindingService;
//@formatter:on

import org.springframework.stereotype.Service;

import java.util.HashMap;
import java.util.Map;

@Service
class DefaultServiceInstanceBindingService implements
    ServiceInstanceBindingService {

    private final ServiceInstanceBindingRepository bindingRepository;

    private final ServiceInstanceRepository instanceRepository;

    @Autowired
    DefaultServiceInstanceBindingService(ServiceInstanceBindingRepository sibr,
        ServiceInstanceRepository sir) {
        this.bindingRepository = sibr;
        this.instanceRepository = sir;
    }

    @Override
    public CreateServiceInstanceBindingResponse createServiceInstanceBinding(
        CreateServiceInstanceBindingRequest request) {
        String bindingId = request.getBindingId();
```

```

{
    ServiceInstanceBinding binding;
    if ((binding = this.bindingRepository.findOne(bindingId)) != null) {
        throw new ServiceInstanceBindingExistsException(
            binding.getServiceInstanceId(), binding.getId());
    }
}

```

❶

```

ServiceInstance serviceInstance = this.instanceRepository.findOne(request
    .getServiceInstanceId());

```

```

String username = serviceInstance.getUsername();
String secretAccessKey = serviceInstance.getSecretAccessKey();
String accessKeyId = serviceInstance.getAccessKeyId();

```

❷

```

Map<String, Object> credentials = new HashMap<>();
credentials.put("bucket", username);
credentials.put("accessKeyId", accessKeyId);
credentials.put("accessKeySecret", secretAccessKey);

```

```

Map<String, Object> resource = request.getBindResource();
String appGuid = String.class.cast(resource.getDefault("app_guid",
    request.getAppGuid()));

```

❸

```

ServiceInstanceBinding binding = new ServiceInstanceBinding(bindingId,
    request.getServiceInstanceId(), null, appGuid);

```

```

this.bindingRepository.save(binding);

```

❹

```

return new CreateServiceInstanceAppBindingResponse()
    .withCredentials(credentials);
}

```

```

@Override
public void deleteServiceInstanceBinding(
    DeleteServiceInstanceBindingRequest request) {
    String bindingId = request.getBindingId();
    if (this.bindingRepository.findOne(bindingId) == null) {
        throw new ServiceInstanceBindingDoesNotExistException(bindingId);
    }
    this.bindingRepository.delete(bindingId);
}
}

```


- ❶ 此方法查找现有服务实例（在 `ServiceInstanceService` 实现中创建）。
- ❷ 返回一个包含对绑定应用程序有用的信息的 `Map <K,V>`。该映射的内容完全是任意的——可以是任何你想要的内容。它将变成一个 JSON 结构，被消费应用程序读取。
- ❸ 准备好之后，将产生的绑定信息记录到数据库中。
- ❹ 返回包含凭证的 `CreateServiceInstanceAppBindingResponse`。



记住，一旦我们在代理上创建了一个 `ServiceInstanceBinding`，`Cloud Controller` 将成为绑定的记录系统，这使得凭证将成为不可变对象，只有重新创建绑定才能改变它。

保护服务代理

`Cloud Controller` 期望 S3 服务代理的 API 通过 HTTP 基本认证的方式来保护。从 `Cloud Controller` 到 S3 代理的每个请求都将包含一个 `Authentication` 标头，该标头会对代理必须验证的用户名和密码进行编码。

S3 代理的 Spring Boot 应用程序使用 Spring Security 来实现 HTTP 基本认证。可以随意重写这个配置，但为了实现该目的，我们使用了方便的 Spring Boot 自动配置属性 `security.user.name` 和 `security.user.password` 来将用户名和密码硬编码为 `admin`。当然，这些可以被环境变量覆盖，例如 `SECURITY_USER_PASSWORD` 和 `SECURITY_USER_NAME`。

这个例子使用一个由用户的内存表示支持的 `AuthenticationManager`。这是类路径中具有 `spring-boot-starter-security` 的几乎所有应用程序的默认行为，但默认是 Spring Boot 应用程序。对于更复杂的服务代理，你可能会想创建一个管理工具，允许平台的运维人员管理存储在代理中的设置，这些设置也许被存储在数据库中。



在创建扩展平台的服务代理时，安全性是一个重要的考虑因素。只有与 `Cloud Controller` 通信才能访问代理的 API，这一点很重要。由于 `Cloud Controller` 只支持 HTTP 基本认证，如果代理的 API 可以通过公共主机或路由访问，那么这将是一个主要的安全漏洞，可能会暴露平台所管理的底层基础设施的内部情况。

就这样，大功告成！应用程序按设计成为了服务代理。我们只需要部署它并把它告诉给 `Cloud Controller`。

部署

现在我们已经创建了 Amazon S3 服务代理，可以通过将其部署到 Cloud Foundry 的发行版来开始测试。服务代理可以被配置为在 Cloud Foundry 的任何发行版上运行。

使用 BOSH 发布

部署自定义 Cloud Foundry 服务代理有多种选择。发布服务代理的流行选择是使用名为 BOSH（BOSH 外壳）的工具。BOSH 是用于管理部署的发布工程工具。它可以使用各种不同的 Cloud Provider Interfaces（CPI）与 Amazon Web Services、Google Compute Engine、OpenStack、vSphere 和 Microsoft Azure 等基础设施即服务（IaaS）平台进行通信。CPI 是 BOSH 架构的可执行组件，用作解释层与 IaaS 提供商公开的 API 进行通信。CPI 采用 IaaS 的专有 API，并将 IaaS 的不同功能转换为 BOSH 中用于描述复杂部署的通用领域语言。BOSH 管理 IaaS 平台上的虚拟资源，并确保实际的系统状态处于管理所描述的状态。通过这种方式，使得 BOSH 与支持融合基础设施的 Puppet 或 Chef 等配置管理工具不同。这些工具试图把现有的节点修改成符合规定的状态。BOSH 支持不可变的基础设施，它每次从虚拟机镜像上重新开始创建。

BOSH 的核心是发布的思想。一次发行需要包含源文件、配置文件、安装脚本等。简而言之，其中包含重复发布给定组件所需的一切。Apache Zookeeper 发行版可能包含 Apache Zookeeper 的源代码、其配置默认值和初始化脚本。

并非所有版本都是一样的。例如，在一个集成测试环境中有两个 Zookeeper 节点，但在生产环境中有几个节点都是可能的。部署清单参数化 BOSH 发布，其描述了需要多少个节点，要传递给节点上每个资源的配置属性等。

BOSH *stemcell* 是基本操作系统镜像。BOSH 官方网站（<http://bosh.io>）上维护了众多的 *stemcell*。

BOSH 非常适合管理复杂系统的部署。如果有什么变化只需要改变部署清单，创建一个新的版本，然后告诉 BOSH 新版本即可。它将根据新的发布重新部署。

为什么我们应该使用 BOSH 发布自定义服务代理，而不是使用 Cloud Foundry 本身部署应用程序？毕竟，在平台上开发应用程序是不是也可以在平台上进行部署？这个问题的简短答案是“是”，这取决于你打算使用自定义代理管理的虚拟化计算资源。

我们的服务代理通过 REST API 与 Amazon Web Services IaaS 直接通信。服务代理不负责创建 S3 实例本身，只负责调用 AWS 的能力。

如果我们的代理只需要使用单个 IaaS 进行编排，那么使用 IaaS 的基于 HTTP 的 API 来

管理其资源是很好的。这与我们使用 Amazon S3 服务代理所做的相似，这时候就没必要创建 BOSH 发布。使用 BOSH 的好处在于，它可以帮助我们管理服务代理发布，而这些发布又需要与 Cloud Foundry 发布的 IaaS 层协调。

在 BOSH 版本中捆绑服务代理的常见原因是，有些代理类型需要管理由大型分布式系统组成的服务实例。这种服务代理需要对 IaaS 层中提供的存储设备和网络资源进行更细粒度的控制。对于我们的 S3 代理，我们不需要为创建 BOSH 发布而担心，因为我们的代理不需要与 IaaS 层进行通信。正因为如此，我们可以使用 Cloud Foundry 的 Cloud Controller API 将代理作为常规应用程序部署。

使用 Cloud Foundry 发布

下面将我们的服务代理部署到 Cloud Foundry。服务代理需要一个绑定为 `s3-service-broker-db` 的 MySQL 实例，所以我们需要在部署应用程序之前先配置一个。如果你使用的是 PCFDev 或 Pivotal Web 服务，则有一个名为 `p-mysql` 的 Pivotal 管理的 MySQL 数据库。Pivotal Web Services 和 PCFDev 的套餐计划是不同的。如果你使用的是 Pivotal Web Services，则可以使用以下方法，你可以通过检查 `cf marketplace` 命令来验证（见示例 14-17）。在写本书时，还有一个名为 ClearDB 的 Pivotal Web Services 产品，它提供了 MySQL。

示例14-17 在Pivotal Web Services上配置一个p-mysql实例

```
cf create-service p-mysql 100mb s3-service-broker-db
```

如果你编译了服务代理，则只需使用与服务代理本身相同的目录中的 `cf push` 将其推送到 Cloud Foundry 即可。你会在那里找到一个 `manifest.yml` 文件，它描述了运行这个应用程序所需的所有东西（见示例 14-18）。

示例14-18 描述如何将服务代理部署到Cloud Foundry的manifest.yml文件

```
---
applications:
- name: s3-service-broker
  path: ./target/s3-service-broker.jar
  buildpack: https://github.com/cloudfoundry/java-buildpack.git
  memory: 1024M
  instances: 1
  timeout: 180
  host: s3-service-broker-${random-word} ❶
  env:
    SPRING_PROFILES_ACTIVE: cloud
    AWS_ACCESS_KEY_ID: replace
    AWS_SECRET_ACCESS_KEY: replace
```



```
services:
- s3-service-broker-db
```

- ❶ `${random-word}` 标记告诉 Cloud Foundry 生成一个随机的单词序列，以避免 URI 冲突。这在像 Pivotal Web Services 这样的多租户环境中特别有用。

你需要自定义这个 `manifest.yml`，为 Amazon Web Services 提供 `AWS_ACCESS_KEY_ID` 和 `AWS_SECRET_ACCESS_KEY` 的值。如果你不希望将这些信息硬编码到 `manifest.yml` 中（我们不怪你），请告诉 Cloud Foundry 在执行 `cf push` 时不要启动服务代理。手动绑定变量，然后启动应用程序（见示例 14-19）。

示例14-19 用于部署和配置Amazon S3服务代理的CLI命令

```
cf push --no-start
cf set-env s3-service-broker AWS_ACCESS_KEY_ID _yourvalue_
cf set-env s3-service-broker AWS_SECRET_ACCESS_KEY _yourvalue_
cf start s3-service-broker
```



需要从 Amazon Web Services 控制台中检索 Amazon Web Services 凭证。请参阅 Amazon Web Services 文档（<http://amzn.to/2vmUv3L>）获取有关该过程的最新信息。

我们的 Amazon S3 服务代理现在在这个平台上完全运行。我们准备通过注册和启用服务代理向其他应用程序公开此服务。

注册 Amazon S3 Service Broker

我们必须注册服务代理。我们需要使用服务代理的 URL。这可以通过 `cf apps` 调用获得（见示例 14-20）。

示例14-20 返回当前空间中Cloud Foundry应用程序的状态

```
→ cf apps
Getting apps in org cloud-native-java / space cnj as cnj@...
OK
```

name	requested state	instances	memory	disk	urls
s3-service-broker	started	1/1	512M	1G	s3-serv..cfapps.io

在我们的例子中，应用程序的 URL 是 `http://s3-service-broker-speckled-swallow.cfapps.io`。你应该记得，我们使用默认的用户名 `admin` 和默认密码 `admin` 配置了服务代理。唯一需要考虑的另一个问题是服务的可见性。如果你将应用程序部署到 Pivotal Web Services（你可能没有多租户环境的管理权限），那么你的服务代理将仅对同一



空间中的其他应用程序可见。你可以通过在 `cf-create-service-broker` 调用上加上 `--space-scoped` 来规定可见范围。另一方面，如果你使用的是 PCFDev，那么就没有必要去处理这些信息。有了这些信息，创建服务代理就很容易（见示例 14-21）。

示例14-21 使用Cloud Foundry注册服务代理

```
cf create-service-broker s3-broker admin admin \
http://s3-service-broker-speckled-swallow.cfapps.io --space-scoped
```

服务代理现在应该可以在 Cloud Foundry 市场中看到（见示例 14-22）。

示例14-22 检查Cloud Foundry市场

```
→ cf marketplace
Getting services from marketplace in org cloud-native-java / space cnj as cnj@...
OK
```

service	plans	description
...		
s3-service-broker	basic	Provides AWS S3 access
...		



现在你有一个服务代理，那么你将能够调配服务的实例，然后将这些实例绑定到应用程序。如果你想最终删除服务代理，则应该从 `cf delete-service-broker` 开始，如果失败了，有两个非常有用的命令可以了解下。如果不再有服务代理为你服务，则 `cf purge-service-offering` 将递归地删除给定服务（或给定服务的特定计划套餐）的所有实例。`cf purge-service-instance` 会强制删除给定服务的实例。

创建 Amazon S3 服务实例

现在，我们的 S3 代理正在运行，并在 PCF Dev 市场上注册为服务。我们可以开始创建服务实例来创建新的 S3 存储桶了，如示例 14-23 所示。

示例14-23 使用新的S3代理创建第一个服务实例

```
$ cf create-service amazon-s3 s3-basic s3-service

Creating service instance s3-service in org pcfdev-org / space pcfdev-space as <?pdf-cr?>admin
OK
```

如果在 `s3-broker` 应用程序中设置为环境变量的 Amazon Web Services 凭证正确，则示例 14-24 中显示的命令将执行成功。如果该命令运行没有错误，则意味着在 Amazon Web Services 账户上创建了新的 Amazon S3 存储桶和相应的 IAM 用户。在我们可以看



到这些资源之前，我们需要为将使用新的 Amazon S3 服务实例的新应用程序部署创建服务绑定。



如果在创建新的 Amazon S3 服务实例后你看到失败信息，这是因为提供的 Amazon Web Services 凭证无效。要解决此问题，请确保该凭证的用户具有用于管理 IAM 用户和管理 Amazon S3 存储桶的正确资源策略。你可以使用身份识别和访问管理工具在 Amazon Web Services 控制台上配置策略。

消费服务实例

当涉及使用新服务扩展平台时，还应将重点放在为开发人员提供在云原生应用程序中使用这些服务的简单方法。虽然平台工程的主要部分是构建服务，但使这些服务尽可能易于使用同样重要。为了有效地做到这一点，我们需要考虑消费这些新服务的工作负载的不同应用运行时环境。

在应用程序中实现功能和将功能作为服务提供之间存在分界线。经验法则是，如果在所有应用程序中都需要实现相同的功能，则应该提供服务。在编写客户端库来消费新服务时，我们也采取同样的法则。对于大多数与服务实例集成的场景，现有的客户端库或 SDK 可用于应用程序。在使用 Amazon S3 服务实例的情况下，我们将使用 Amazon Web Services Java SDK 与 Amazon S3 的存储 API 进行交互。

在开发 JVM 应用程序以在 Cloud Foundry 上使用自定义服务时，平台工程师应该对应用程序开发人员如何使用这些服务持有自己的观点。做一名“自以为是”的平台工程师还有一个额外的好处：你可以让架构审查委员会不必担心应用程序的源代码符合规定标准的程度。我们可以通过对应用程序框架如何与平台集成进行自我评估来实现应用程序合规性的大部分自动化工作。借助 Spring Boot，我们可以采用与应用程序框架开发人员相同的观点，通过提供 Spring Boot starter 项目，自动使用我们通过扩展 Cloud Foundry 提供的服务实例。Spring Boot 的自动配置非常适合此目的，我们知道，Spring Boot 应用程序都需要与我们的 S3 存储桶进行通信，因此我们可以在一次自动配置中描述集成的中心部分，并在任何地方重复使用。

为了使应用程序开发人员能够与我们的 Amazon S3 代理通信，我们已经为 Amazon Web Services SDK 的 AmazonS3 客户端创建了一个自动配置。

AmazonS3 依赖于一个名为 `BasicSessionCredentials` 的类型，而这个类型又需要来自 `Credentials` 对象的值。如果你指定了正确的属性（`amazon.aws.access-key-id` 和 `amazon.aws.access-key-secret`），则所有这些都会为你配置好的（使用 AWS 中的安全令牌服务工具来减小给定令牌的范围），如示例 14-24 所示。



示例14-24 使用新的S3服务代理创建第一个服务实例

```
package amazon.s3;
```

```
import com.amazonaws.auth.AWSCredentials;  
import com.amazonaws.auth.AWSCredentialsProvider;  
import com.amazonaws.auth.BasicAWSCredentials;  
import com.amazonaws.auth.STSSessionCredentialsProvider;  
import com.amazonaws.regions.Regions;  
import com.amazonaws.services.s3.AmazonS3;  
import com.amazonaws.services.s3.AmazonS3ClientBuilder;  
import com.amazonaws.services.securitytoken.AWSSecurityTokenService;  
import com.amazonaws.services.securitytoken.AWSSecurityTokenServiceClient;  
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;  
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;  
import org.springframework.boot.context.properties.EnableConfigurationProperties;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
```

```
@EnableConfigurationProperties(AmazonProperties.class)
```

```
public class S3AutoConfiguration {
```

❶

```
@Bean
```

```
@ConditionalOnMissingBean(AmazonS3.class)
```

```
@ConditionalOnClass(AmazonS3.class)
```

```
public AmazonS3 amazonS3(AmazonProperties awsProps) { ❷
```

```
    String rootAwsAccessKeyId = awsProps.getAws().getAccessKeyId();
```

```
    String rootAwsAccessKeySecret = awsProps.getAws().getAccessKeySecret();
```

```
    AWSCredentials credentials = new BasicAWSCredentials(rootAwsAccessKeyId,  
        rootAwsAccessKeySecret);
```

```
    AWSSecurityTokenService stsClient = new AWSSecurityTokenServiceClient(  
        credentials);
```

```
    AWSCredentialsProvider credentialsProvider = new STSSessionCredentialsProvider(  
        stsClient);
```

```
    return AmazonS3ClientBuilder.standard().withRegion(Regions.US_EAST_1)  
        .withCredentials(credentialsProvider).build();
```

```
    }
```

```
}
```

- ❶ 当且仅当在应用程序上下文中没有相同类型的现有 bean 时，自动配置才会提供一个 bean。
- ❷ 配置取决于自定义配置属性组件 AmazonProperties，以解析 Amazon Web Services 访问令牌、密码和令牌持续时间。



消费该服务的应用程序只需要将此自动配置添加到类路径中，并为两个属性 `amazon.aws.access-key-id` 和 `amazon.aws.access-key-secret` 提供值即可。

S3 客户端应用程序

我们来看一个简单的 S3 客户端 REST API。这是一个 Spring Boot 应用程序，它只依赖于我们的自动配置和 `spring-boot-starter-web`。当它被部署到 Cloud Foundry 时，我们创建一个 S3 服务实例（为其分配一个逻辑名称 `s3-service`），并将其绑定到我们的 S3 客户端（见示例 14-25）。

示例14-25 部署S3客户端

```
cf create-service s3-service-broker basic s3-service
cf push
```

该应用程序是一个具有两个端点的 REST 服务：支持 HTTP GET 请求的 `s3/resources` 和支持多个文件上传的 POST 请求的 `/s3/resources/{name}`。上传一个文件，然后检查它是否存在，示例 14-26 给出了代码。

示例14-26 S3RestController与配置好的S3服务实例配合使用

```
package com.example;
```

```
import amazon.s3.AmazonProperties;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.model.*;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.hateoas.Link;
import org.springframework.hateoas.Resource;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;
```

```
import java.net.URI;
import java.time.Instant;
import java.util.List;
import java.util.stream.Collectors;
```

```
@RestController
@RequestMapping("/s3")
class S3RestController {
```

```
    private Log log = LogFactory.getLog(getClass());
```




```
private final AmazonS3 amazonS3Client;

private final String defaultBucket;

@Autowired
public S3RestController(AmazonProperties amazonProperties,
    AmazonS3 amazonS3Client) {
    this.amazonS3Client = amazonS3Client;
    this.defaultBucket = amazonProperties.getS3().getDefaultBucket();
    log.debug("defaultBucket = " + this.defaultBucket);
}
```

```
❶
@PutMapping("/resources/{name}")
ResponseBody<?> upload(@PathVariable String name,
    @RequestParam MultipartFile file) throws Throwable {

    if (!file.isEmpty()) {
        ObjectMetadata objectMetadata = new ObjectMetadata();
        objectMetadata.setContentType(file.getContentType());
        PutObjectRequest request = new PutObjectRequest(this.defaultBucket, name,
            file.getInputStream(), objectMetadata)
            .withCannedAcl(CannedAccessControlList.PublicRead);
        PutObjectResult objectResult = this.amazonS3Client.putObject(request);
        URI location = URI.create(urlFor(this.defaultBucket, name));
        String str = String
            .format("uploaded %s at %s to %s", objectResult.getContentMd5(), Instant
                .now().toString(), location.toString());
        log.debug(str);
        return ResponseEntity.created(location).build();
    }
    return ResponseEntity.badRequest().build();
}
```

```
❷
@GetMapping("/resources")
List<Resource<S3ObjectSummary>> list() {

    ListObjectsRequest request = new ListObjectsRequest()
        .withBucketName(this.defaultBucket);

    ObjectListing listing = this.amazonS3Client.listObjects(request);

    return listing.getObjectSummaries().stream().map(this::from)
        .collect(Collectors.toList());
}
```



```

private String urlFor(String bucket, String file) {
    return String.format("https://s3.amazonaws.com/%s/%s", bucket, file);
}

private Resource<S3ObjectSummary> from(S3ObjectSummary a) {
    Link link = new Link(this.urlFor(a.getBucketName(), a.getKey()))
        .withRel("location");

    return new Resource<>(a, link);
}
}

```

❶ 该端点支持文件上传。你可以使用 curl 或 PostIt 浏览器插件来尝试。

❷ 该端点枚举 Amazon S3 实例中所有上传的资源。

S3RestController 使用配置好的 AmazonS3 客户端与 S3 通信。自动配置需要一个有效的访问密钥和密码。我们可以将它们映射到配置属性中的 S3 支持服务环境变量上。

示例14-27 具有两个配置文件的示例应用程序的配置，一个用于本地开发，另一个配置在Cloud Foundry上，且云配置文件处于活动状态

```

spring.profiles.active: development
spring:
  http:
    multipart:
      file-size-threshold: 10MB
      max-file-size: 10MB
      max-request-size: 10MB
server:
  port: 8081
---
spring:
  profiles: cloud
  application:
    name: ${vcap.application.name:s3-sample}
amazon:
  aws:
    ❶
    access-key-id: ${vcap.services.s3-service.credentials.accessKeyId}
    access-key-secret: ${vcap.services.s3-service.credentials.accessKeySecret}
  s3:
    default-bucket: ${vcap.services.s3-service.credentials.bucket:bucket}
---
spring:
  profiles: development

```




```

application:
  name: s3-sample-app
amazon:
  aws:
    access-key-id: replace
    access-key-secret: replace

```

- ① 后台服务凭证以名为 `VCAP_SERVICES` 的环境变量存储在 JSON 结构中。Spring Boot 将 JSON 结构扁平化为 Spring Environment 中的键，以 `vcap.services.service-id` 开始，其中 `service-id` 是绑定到应用程序的服务的逻辑名称。在该示例中，我们创建了一个名为 `s3-service` 的 S3 服务实例。此配置将访问密钥、密码以及存储桶信息解引用，以配置 AmazonS3 客户端。

运行测试

随着服务代理和客户端应用程序的部署完成，我们的解决方案也大功告成。首先，我们通过向 `/s3/resources/{file-name}` 发出一个 PUT 请求来写一个文件到 S3，其中 `file-name` 是文件的名字，稍后会引用它（见示例 14-28）。

示例14-28 使用名称 `test` 写入 Amazon S3（我们使用省略号缩短此列表的行长度——使用已部署的 Amazon S3 客户端的 HTTP URI 替代）

```
curl -v -F file=@/path/to/an/img.png http://s3-sa...cfapps.io/s3/resources/test
```

你应该能够通过向 `/s3/resources` 端点发出 GET 请求来确认结果（见示例 14-29）。

示例14-29 从 Amazon S3 读取

```
curl -v http://s3-sample-app-...cfapps.io/s3/resources
```

总结

在本章中，我们了解了服务代理，这是 Cloud Foundry 平台的一个基本组成部分。服务代理为我们提供了统一的服务提供和消费接口，简化了运维工作。Spring Boot 的自动配置是一个很好的补充：它为应用程序开发人员提供了一种无摩擦的插入配置服务的方法。服务代理和自动配置最重要的一点是，它们对一致性进行了优化，从而提高了开发速度。

强大的服务代理范式诞生于 Cloud Foundry，并且在其他平台上被采用。Open Service Broker API (<https://www.cloudfoundry.org/open-service-broker-api-launches-as-industry-standard/>) 是一项旨在建立跨云提供服务交付标准化的举措，将服务代理带到其他云供应商，包括 Google、Red Hat 和 IBM 等。



第 15 章

持续交付

在前面章节中，我们着重介绍了如何构建有生产价值的软件，并探讨了软件的可交付性。我们也广泛研究了 Cloud Foundry。引用 Andrew Clay Shafer 的话说，Cloud Foundry 是“武装的基础架构”，或者是开发人员开箱即用的基础架构，其针对部署和运行应用程序进行了优化。我们已经了解了如何通过 Cloud Foundry 获得应用程序，以及如何构建一旦部署就可以充分利用平台特性的应用程序。

本章我们将讨论 Cloud Foundry 和微服务架构的组织动机。我们将来认识持续交付的实践和 Concourse 应用，Concourse 是一种支持持续交付管道的技术。

持续集成之外

让我们退一步，再看一下从软件开发到软件部署和发布的过程。Spring Boot 使得构建有生产价值的服务变得十分容易。它减少了对软件构建认知（和实际）的投资，以获得有效的软件。它使组织能够更快地实现结果。许多组织也使用 CI（持续集成）来为开发人员提供更快的代码和测试反馈。如果使用 CI、Spring Boot、测试驱动开发和敏捷软件开发方法学，以及无数的 ITIL 单据，且要花费几天、几周甚至几个月的时间才能实现硬件配置和基础设施部署，那么这些方法学又有什么用呢？如果从概念到生产的工作流程中， workflow 测试、运维等任何后续阶段都成为瓶颈，那么在编写代码时需要做什么优化呢？

没有人在构建完了微服务后将其手动部署到 ITIL 请求的融合 WebSphere 实例上，而这些实例还是在由这些请求调配的硬件和基础架构上运行的。这至多是 SOA（面向服务的体系结构），而不是微服务。微服务支持敏捷。这是其价值流的一部分，价值流中的每一步都有助于交付软件。不要在接受微服务时而忽略了这个更大的上下文。

IT 提供软件的速度比我们想象中的更快！

——从来就没有商业人士

大多数组织的价值流如下所示：企业有一个商业的构思，然后要求产品管理部门、用户体验部门以及 IT 部门来实现，并且希望在不太遥远的未来，这些成果也能让客户看到。客户是可能愿意支持我们或为我们的努力付费的人，可能是内部用户或外部付费客户。客户手中的软件应该代表你的业务或组织的价值。如果我们能够自然而然地简化和加速价值的流动，业务部门就会喜欢它。但 IT 部门的传统姿态是，如果我们行动速度太快，就会引入错误。这意味着大部分流程（交付流程）都是手动的，这是问题的实质。

你的交付管道是手动的吗？需要几十人做代码筛选和烦琐的错误检查吗？部署是否需要手动，需要手动向生产环境添加文档驱动的增量吗？部署很少发生——一个月一次，还是每三个月一次，还是每两年一次？部署日的到来是否令人恐惧？部署时是否需要人员登录桥接线？是否需要每个团队和相关人员联合组建作战室？如果满足其中任何一点，那么这就是一个破坏的交付管道，这个组织的功能也是不正常的。

持续交付提供了一个更好的方式。持续交付作为一种追求，并不是什么新事物。它的初衷很简单：尽可能自动化价值流中的所有部分。特别是，在代码被提交之后自动化所有的东西，这样从提交到生产的路径是可重复的，无须人工干预。几十年来，一些组织已经这样做了，或者做法很类似。

持续交付如今的实践，借鉴了精益生产的理念：确保高质量产品的快速交付，重点在于消除浪费和降低成本。这样可以节省成本和资源，提高产品质量，缩短产品上市周期。

这可能看起来似是而非，但去除流程中烦琐的手动部分可以减少缺陷，提高质量。像软件部署这样的事情是高度技术性的，但却是劳动密集型的，是一种乏味的工作。这种组合充满隐患，人类不是专门为它而工作的。错误和不一致必然会出现。自动化一切——在此过程中发现错误。这样交付过程对于阶段构建和生产而言是一样的。如果在这个过程中有错误，那么可以在阶段构建中解决，从而避免调试生产中出现错误。

软件形式的自动化实际上是构建版本可控的工件，以集中和社交化部署的方式工作。不一定是部署专家，任何人都可以从版本控制中获取并运行软件来构建可靠的应用程序版本。随着交付速度的提高，修复缺陷的成本降低了，因为新版本可以很快完成。速度降低了变更的成本，这使得尝试变更变得更容易。变化是企业生存之本。

做对了，软件交付应该像在版本控制系统中选择一个版本一样简单，然后按下一个绿色的“部署！”按钮。有些组织更进一步，去掉了“部署！”按钮并践行了持续发布，`git push`将直接触发部署到生产的执行，它会假设所有质量测试都已通过。

John Allspaw 在 Flickr 以及后来的 Etsy

John Allspaw 帮助 Flickr（以及后来的 Etsy）接受了持续交付（Continuous Delivery）

的理念并发布了一个巨大的单体应用程序。Etsy 的故事详见 *Network Wolrd* (<http://bit.ly/2vnAJ8k>) 中的一篇文章，其在过去的十年中已经成为很多组织的灵感来源。Allspaw 长期以来一直推崇持续交付，并且讲了为了贯彻持续交付，他的组织在文化和技术上所做的事情。Flickr 和 Etsy 都是在线提供的软件即服务。Allspaw 谈了很多有关 Web 应用程序的交付（怀疑本书的大多数读者将会用到）改变了交付方式方面的内容（例如，2009 年的 Velocity 会议，<http://bit.ly/2rrFnPl>）。

在 SaaS 模型中，没有以前版本的软件。刷新浏览器，你将获得最新版本的代码。大多数组织使用分支的方式来处理软件交付中的回归或缺陷，这种方式已经不再适用。在 Etsy，所有的变更都在 master 上发布。如果需要引入代码库的功能，那就使用新的功能标志。这些功能标志允许 Etsy 在一定比例的受众中部署功能。这些功能可以存储在代码库中，收集信息并身处实际请求进入服务的热路径中，然后生产可以监视系统的负载。这些信息再为容量规划和架构讨论提供参考。当企业宣布一个功能，并“发布”，这就将成为一个顽固的事情，该服务已经连续数周或数月快速运行。如果某些内容出错，则只需使用功能标志禁用该功能即可向前回滚。

Etsy 很快学会了如何自动化部署代码到生产中，并且在软件和组织本身上构建保护措施来支持持续交付。他们没有提出一天到底有多少次部署的目标，他们开始可以频繁地安全地部署到生产。

如果你一天不烦躁 10 次，你就不能一天部署 10 次。

——John Allspaw

Netflix 的 Adrian Cockroft

从 2009 年起，Adrian Cockroft 开始帮助 Netflix 从单体的数据库和应用程序架构转型到云原生架构。在众多的见解（以及你在阅读本书时看到的那些伟大的开源技术）中，从 Netflix 进军微服务的角度来看，迭代速度比效率更有价值。组织走得越快，某个服务速度比设想的慢就越不那么重要，因为这有助于组织保持竞争优势。它控制着这个领域。Adrian 在众多的采访中都谈到了这一点，比如这个 Scale (<http://bit.ly/2s9NRvY>)：

关于团队变得敏捷后的规模有一个限制。这个限制可能是由 Etsy 目前正在做的事情来定义的，因为他们正是运行一个非常敏捷的单体应用程序的最好的例子。这是一个惊人的壮举，但是大多数人都说：“这太棒了，但是用更小的团队做事情会更容易。效率不高，但我更关心速度而不是效率。”

——Adrian Cockroft

唯快不破，那么你迭代的速度呢？

亚马逊的持续交付

在 2015 年 AWS re:Invent 大会上，Rob Brigham 在他的演讲“亚马逊的 DevOps：我们的工具和流程”（<https://www.youtube.com/watch?v=esE FaY0FDKc>）中，讲述了一个引人注目的关于亚马逊如何从巨石应用转向微服务的故事。

Brigham 解释说，对于亚马逊来说，DevOps 在效率方面有所提高，可以减少软件交付生命周期中闲置的时间。从表面上看，这可能不是你一直在寻找的 DevOps 的奥义。但是对于 Brigham 来说，这个陈述改变了整个计算世界，就像我们所知道的那样。

他表示，亚马逊的交付生命周期分为两个方面：开发者和客户。为了让客户从开发者那里获得新功能，每一次变化都将经过构建、测试和发布阶段。构建的版本发布后，开发人员可以监控客户如何与新功能交互。这个反馈循环是非常重要的，因为它为开发人员提供了所需的洞察，从而确定他们接下来要做什么。

Brigham 表示，该反馈回路可以让亚马逊的开发人员不再通过猜测来确定客户更喜欢什么样的功能。他还表示，亚马逊会尽可能快地完成一个反馈循环。亚马逊的持续交付源于减少开发人员花费在构建、测试和发布代码上的时间，因为这项工作客户是看不见的。通过最大限度地减少在交付阶段花费的时间，持续交付可缩短客户看到最终产品的时间。

流水线

当我们谈论持续交付时，其实是在谈自动化的流程——流水线，它将软件交付到生产环境。流水线大体上是指软件从持续集成一直到生产这个过程。也有人叫这个过程是“代码兑现”。

这个自动化过程被称为交付流水线。流水线对客户价值传递的所有方面都是透明的。业务部门和 IT 部门可以就价值流中分别承担多少工作进行讨论。它就像是一个漏斗，所有的变化都会通过它，如热点修补、新功能、回归测试等。开发和运维都可以进入漏斗。没有人能做逾越这个漏斗的操作是很关键的！如果流水线不能满足所有相关方的要求，那么它就不可信，所以必须囊括所有要求。流水线是单个可见事物，在那里各方反馈都是可见的，以便出现任何错误都可以尽早解决。

每条流水线都有一系列的基本步骤，在将源代码部署到生产环境之前，需要对源代码进行更改，然后进行安全构建和测试（就像传统的持续集成一样）。流水线中的步骤本身应该在受版本控制的工件中描述。该工件可能是更多的软件、脚本或配置。

持续集成是有价值的，因为它会迫使团队让软件可以在他们的机器之外的某个地方工作。如果在运行代码时出现任何问题，则必须捕获这些问题，并将其添加到版本控制和自动部署中。这应该是新团队在首次迭代中设置持续交付流水线的首要任务。

在持续交付中，对软件的更改（通常以提交给 Git 等版本控制系统的代码主分支的形式）通过流水线进行。版本控制系统应该可以重现应用程序及其环境所需的一切。除了生产数据以外，所有的东西都应该在版本控制中，甚至是流水线定义本身。对于在 Cloud Foundry 上运行的现代 Spring 应用程序，这可能意味着使用 Maven 或 Gradle 构建，在 Cloud Foundry manifest.yml 以及 Spring Cloud Config Server 中做配置的修订。

大多数交付流水线将执行以下步骤（可能更多）：构建和单元测试，集成测试，发布和部署。

构建和单元测试

在这个步骤中，软件会以后续步骤中可以使用的形式被打包或编译。重要的是，除此之外，对源代码的需求很少（除了代码质量分析之外）。随着应用程序从一个环境迁移到另一个环境，应该使用相同的二进制或打包代码。像 JFrog 的 Artifactory 这样的工件存储库可以用来存储成功的产品。

集成

在这一步中，将来自上一步的工件部署到生产环境中，然后执行。Cloud Foundry 通过简洁的 .yml 清单文件来部署相同且可重复的环境变得十分简单。在这里，测试可能会更详尽，并且会测试端到端的行为。也可以在冒烟测试中验证服务依赖关系的状态。冒烟测试是一种软件测试，旨在确保应用程序最重要的功能可以正常工作。也可以让人类做一些探索性测试。一旦这个步骤通过，软件就可以转到发布步骤。

发布

在这个步骤中，工件会被标记为可发布的。不一定必须发布。它可能只是在一个工具存放处放着。重要的是我们已经有信心把它部署到生产。如果是自动部署的，那就是持续部署。如果没有自动发布，那就是持续交付。

部署

在这一步中，可发布的软件被提升到生产环境。在线 SaaS 应用通常使用蓝绿部署等模式来实现零停机时间部署。

测试

停止依赖检测的方式来满足质量。而是应该将产品质量建设放在第一位，融入到产品中来消除对大规模检测的需求。

—— W. Edwards Deming, “全面质量管理 14 点”

在安全的情况下更快的速度才有意义。如果你说：“放弃所有的质量控制，我们会走得更快！速度！但没有质量！来吧！”那么没人会相信你说的话。

随着变更在流水线中移动，必须通过逐渐详尽（和更慢）的质量把控。无论何时变更导致流水线失败，构建失败，那么恢复构建必须是参与团队的首要任务。直到流水线再次健康之前，流水线中不再接受新的变更。丰田公司坚信，生产上出现任何缺陷都应该停止生产线，直到缺陷得到修复。我们也必须停止流水线。

持续交付的前提是把测试作为质量把控之法。我们详细地讨论了测试，因为它适用于 Spring 应用程序。一般来说，需要考虑四种测试。如果你转向持续交付，你应该尽可能利用这些测试的优势。Brian Marick 描绘了如图 15-1 所示的测试象限。

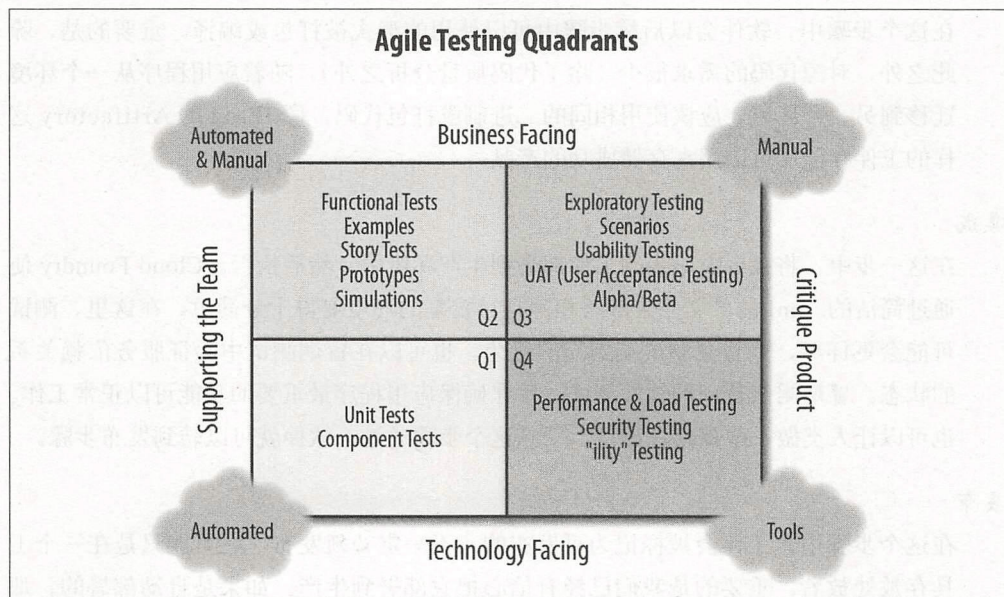


图15-1 Brian Marick的测试象限

持续交付微服务

在软件设计中要考虑的最重要一点就是模块化。如果我们从机械意义上考虑模块化，则系统的组件被设计成模块，可以在发生机械故障时替换它。例如，在汽车发动机中，如果某个火花塞失效，则不需要更换整个发动机。然而，在软件中，一个小小的改动就需要替换整个应用程序。正因为如此，我们应该感到幸运，我们是在构建软件而不是在造车。今天最有价值的公司是使用位和字节而不是塑料和金属来创建。但是，尽管取得了这些进展，仍然有一些汽车公司可以比发布软件更快地发布汽车。

为什么更换汽车上的轮胎更容易，而不是更换软件应用程序的模块？

模块化还可以为开发人员提供应用程序的功能关系图。通过可视化和映射由应用程序源代码编排的复杂流程，开发人员和架构师都可以更容易和精确地查看在哪里更改软件。

应用程序的源代码是一个有很多连续的位和字节的系统，它总是在不断发展——一个又一个的变化。但是，随着系统的源代码的扩展或协议需要，软件的一些很小的变化就会要求我们构建和部署整个应用程序。为了将细小的变更部署到生产环境，我们需要部署其他所有我们没有更改的东西。

在共享部署流水线上将变更部署到生产就像是在一辆开往“生产市”的公交车上购买单程票。在大城市搭巴士是比开自己车成本低廉的替代方案。问题是，你不能完全自主选择何时上车。部署管道时也是如此。

不必等待下一班车，如果你准备出发就可以召到公共汽车呢？这不是一个新的想法！实际上，这个想法改变了我们使用技术来按需用车旅行的方式。现在，如果我们将这个想法应用于生产变更呢？

持续交付就像使用按需乘车的共享服务一样将你的更改按需部署到生产。

当团队共享一个应用程序的部署流水线时，他们被迫按照一个计划被统一安排，他们很少或根本无法控制整个流水线。出于这个原因，创新被扼杀了，因为人们必须等到下一班车才能得到其有关变更的反馈。

使用微服务的结果是将变更推向生产的途径增多了。

工具

有几个工具可以用来做持续交付。像 Jenkins 这样的工具仍广泛用于基于云的部署。虽然在云计算时代有一些新的发布管理工具，但是许多人还是习惯使用他们熟悉的工具。较新的工具侧重于软件交付生命周期的不同阶段。在本章中，我们将使用名为 *Concourse* 的工具来探索云原生中的持续交付。

Concourse

Concourse 是为云原生而生的工具。Concourse 工具链由 Pivotal 创建。这是一个持续集成工具，允许你使用 Docker 容器来组成简单或复杂的部署流水线。Concourse 与 Jenkins 或 Travis 等工具相比有两个不同点：容器和流水线。

容器

Docker 彻底改变了我们在云中打包和分发应用程序的方式。可以说 Docker 是一个开源的打包工具，它允许你描述和构建一个基于 Linux 的容器来作为应用程序的执行环境。你可以把该 Linux 容器当作一个虚拟机，但实际上它只是一个共享 Linux 内核的文件系统。

Cloud Foundry 在很大程度上使用 buildpack 将容器抽象化，这将智能地在容器内构建你的部署工件，而无须描述执行环境。虽然 Cloud Foundry 使用自己的开源容器管理工具，但 Concourse 允许你使用 Docker 容器来编写流水线。

Docker 是在可组合的部署管道中运行任务的理想选择。Concourse 允许你为在 Docker 容器内执行任务的流水线创建临时的构建、测试和部署环境。

持续交付微服务

我们将构建两个持续交付流水线，这些流水线会将源代码部署至生产中。交付流水线的主要部分是测试。因此，我们将重用关于测试的代码示例（参见第 4 章内容），使用 Concourse 来构建、测试和部署 Spring Boot 微服务。第 4 章提供了两个微服务：Account 微服务和 User 微服务（见图 15-2）。

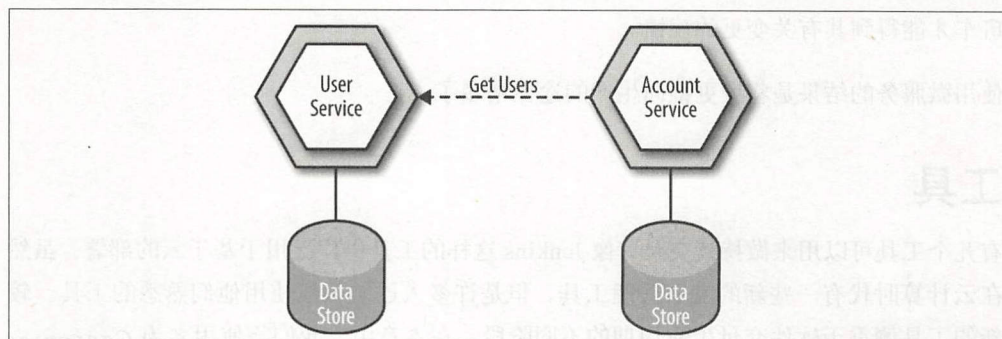


图15-2 Account微服务依赖于User微服务

使用微服务的好处是不同团队可以独立地将功能部署到生产环境中。不用共享部署流水线是这种架构的主要优点。但是新增加了成本：责任。团队必须确保他们的消费者不会中断。这就是持续集成的话题了。在将变更转移到生产之前，通过持续集成尽早且经常地测试我们的微服务。

比起不破坏主要功能的轻微缺陷来说重大变更可能是灾难性的（在数据不一致的情况下）。微服务的每个消费者都需要维护他们所依赖的服务的期望。我们将设计一个执行

持续集成的持续交付流水线，以确保消费者和生产者之间的期望保持不变，然后才能进入生产。

安装 Concourse

安装 Concourse 有很多种方法。最快的方法是使用 Vagrant。有关使用 Vagrant 安装 Concourse 的最新说明，请参见 Concourse 文档 (<http://concourse.ci/vagrant.html>)。

Concourse 有一个名为 fly 的 CLI 工具和一个用于可视化流水线状态的 Web 界面。一旦你部署并运行了 Concourse 并安装了 fly 工具，你就可以将目标设置为本地 Vagrant 安装，并开始创建流水线。

要开始使用 fly，需要先登录到你的 Concourse 实例。如果你使用的是 Vagrant，则 URL 将如下所示：

```
fly -t lite login -c http://192.168.100.4:8080
```

一旦使用 fly 成功登录到 Concourse 实例，就可以开始创建流水线。这里有两个简单的命令：set-pipeline 和 destroy-pipeline。在示例 15-1 中，使用了 set-pipeline 命令。

示例15-1 使用set-pipeline命令创建一个新的流水线

```
fly -t lite set-pipeline -p account-microservice -c pipeline.yml \
-l .pipeline-config.yml
```

这里我们为 Account 微服务创建一个新的流水线。set-pipeline 命令有几个参数，如表 15-1 所示。

表15-1 set-pipeline命令的参数

标志	参数	说明
-p	account-microservice	希望创建的Concourse流水线的名称
-c	pipeline.yml	描述新的Concourse流水线的YAML文件的路径
-l	.pipeline-config.yml	包含密码的YAML文件的路径，这些密码将作为新流水线的参数

请注意，这里的参数值来自本章附带的源代码示例。执行此命令后，你将能够导航到 Concourse admin web 界面，并可以看到准备开始运行的新流水线。

另一个有用的命令是 destroy-pipeline。流水线是不可改变的，这意味着你可以在合适的时候销毁它们并重新创建流水线。我们在示例 15-2 中销毁流水线。

示例15-2 使用destroy-pipeline命令销毁现有的流水线

```
fly -t lite destroy-pipeline -p account-microservice
```

可以看到，我们在这里销毁了在示例 15-1 中创建的流水线。

基本的管道设计

持续交付流水线的基本设计包括构建、测试和部署阶段（如图 15-3 所示）。

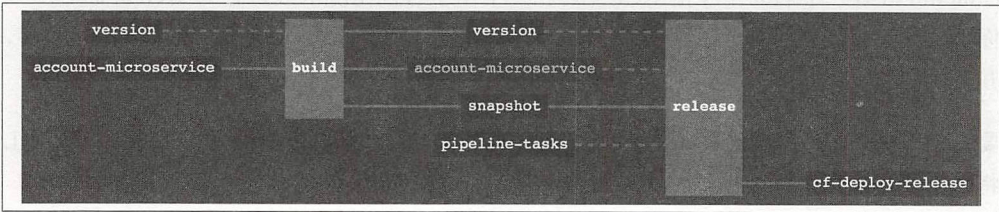


图15-3 Account微服务的流水线

在图 15-3 中，我们可以看到 Account 微服务的 Concourse 流水线定义。Concourse 流水线有三个基本元素：资源（Resource）、任务（Task）和作业（Job）（如表 15-2 所示）。

表15-2 Concourse流水线的基本元素

类型	功能
Resource	资源是任务的输入或输出
Task	任务属于作业，使用 Docker 容器内的资源构建输出
Job	作业将一组任务组合在一起，是流水线中最大的组成部分

你可能已经注意到，在图 15-3 中，此流水线没有集成阶段。Account 微服务还没有任何消费者，但它依赖于 User 微服务。为此，我们将设计一个具有构建、测试和发布阶段的简单流水线。

在表 15-3 中，你可以在 Account 微服务的 Concourse 流水线中找到每项作业的描述。Concourse 支持多种资源类型，可以在你的流水线描述文件中定义。在与本章相关的示例项目中，你会发现 Account 微服务的源代码的目录中包含一个 Concourse 流水线完整定义。

表15-3 Account微服务的Concourse作业

作业	输入	输出	说明
Build	当前版本、Git 源	发布工件、递增版本	Build 作业将克隆和构建 Account 微服务的源代码，如果通过单元测试，将准备一个新的版本发布工件
Release	递增版本、Git 源、发布工件	Cloud Foundry 部署	Release 作业将发布工件并将新版本部署到 Cloud Foundry

图 15-4 显示了将输入到 set-pipeline 命令中的文件，该命令将创建 Account 微服务的 Concourse 流水线。我们要讨论的第一个文件是 pipeline.yml 文件，其中包含了 Concourse 流水线定义。

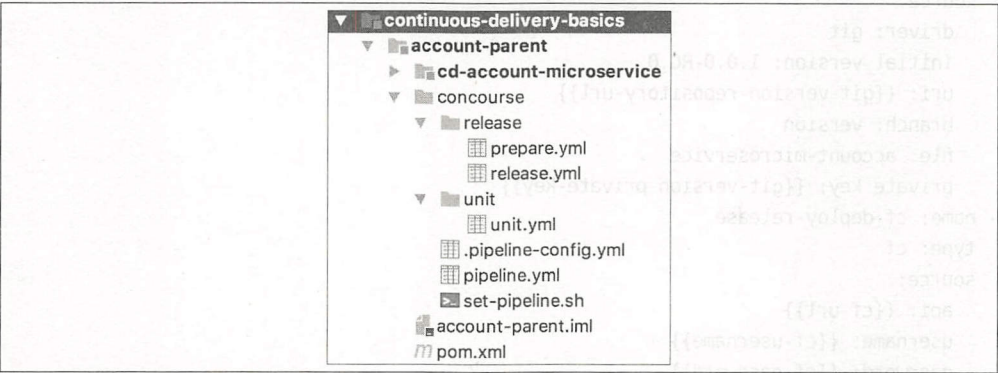


图15-4 Account微服务的源代码目录

在示例 15-3 中，给出了 Account 微服务的 Concourse 流水线定义的第一部分。这些是 Concourse 资源，将是每项作业的输入和输出。在 pipeline.yml 的下一部分，我们将创建使用示例 15-3 中定义的资源作业定义。

示例15-3 在pipeline.yml中为Account微服务定义的资源

```
# Concourse pipeline definition for the account service.
---
resource_types:
- name: maven-resource
  type: docker-image
  source:
    repository: patrickcrocker/maven-resource
    tag: latest
resources:
- name: account-microservice
  type: git
```



```

source:
  uri: {{git-source-repository-url}} ❶
  branch: master
  paths:
    - ./continuous-delivery-basics/account-parent/cd-account-microservice
- name: snapshot
  type: maven-resource
  source:
    url: {{artifactory-url}}
    artifact: cnj:cd-account-microservice:jar
    username: {{artifactory-username}}
    password: {{artifactory-password}}
- name: version
  type: semver
  source:
    driver: git
    initial_version: 1.0.0-RC.0
    uri: {{git-version-repository-url}}
    branch: version
    file: account-microservice
    private_key: {{git-version-private-key}}
- name: cf-deploy-release
  type: cf
  source:
    api: {{cf-url}}
    username: {{cf-username}}
    password: {{cf-password}}
    organization: {{cf-org}}
    space: {{cf-space}}
    skip_cert_check: false
- name: pipeline-tasks
  type: git
  source:
    uri: https://github.com/Pivotal-Field-Engineering/pipeline-tasks
    branch: master

```

❶ 花括号表示我们可以使用 fly set-pipeline 注入的参数和密码。

在示例 15-4 中，给出了 Account 微服务流水线中的作业和任务的定义。在 build 作业中，我们首先克隆 Account 微服务的源代码，并将其作为 unit 任务的输入。现在我们来了解一下 unit 任务，它将使用从 Git 资源克隆的源代码（见示例 15-5）构建和单元测试 Account 微服务。

示例15-4 在pipeline.yml中定义的Account微服务的作业和任务

```
jobs:
- name: build
  max_in_flight: 1
  plan:
  - get: account-microservice
    trigger: true
  - task: unit
    file: account-microservice/concourse/unit/unit.yml
  - get: version
    params: { pre: RC }
  - put: snapshot
    params:
      file: release/cd-account-microservice.jar
      pom_file: account-microservice/source/pom.xml
      version_file: version/version
  - put: version
    params: { file: version/version }
- name: release
  serial: true
  plan:
  - get: snapshot
    trigger: true
    passed: [build]
  - get: account-microservice
    passed: [build]
  - get: version
    passed: [build]
  - get: pipeline-tasks
  - task: prepare-manifest
    file: account-microservice/concourse/release/prepare.yml
    params:
      MF_PATH: ../release-output/cd-account-microservice.jar
      MF_BUILDPACK: java_buildpack
  - task: prepare-release
    file: account-microservice/concourse/release/release.yml
  - put: cf-deploy-release
    params:
      manifest: task-output/manifest.yml
```

示例15-5 unit任务将建立和测试Account微服务

```
# This task will build and run the unit tests specified in the source code
# of the microservice.
---
platform: linux
image_resource:
```



```

type: docker-image
source:
  repository: maven
  tag: alpine
inputs:
- name: account-microservice
outputs:
- name: release
run:
  path: sh
  args:
  - -exc
  - |
    cd account-microservice/source \
    && mvn clean package \
    && mv target/cd-account-microservice.jar \
    ../release/cd-account-microservice.jar

```

在示例 15-5 中，给出了构建和单元测试 Account 微服务的 unit 任务的定义。每个任务都会挂载输入和输出卷在 Docker 容器中运行，这使得我们可以将状态传递给流水线中的每个任务和作业。在这个例子中，我们只在测试通过时才构建 Account 微服务的 JAR 工件。这意味着 JAR 工件将被输入下一个任务中，准备快照构建并将其部署到 Maven 快照存储库。

你创建的每个流水线都可能拥有包含登录名和密码的参数。当创建一个新的流水线时，你可以提供一个配置文件作为输入参数。示例项目的 account-parent/concourse 目录中包含了一个 set-pipeline.sh 脚本。在示例 15-6 中，我们可以看到该 Account 微服务脚本的内容。

示例15-6 set-pipeline.sh文件创建Account微服务流水线

```

#!/usr/bin/env bash

fly -t lite set-pipeline -p account-microservice -c pipeline.yml \
-l .pipeline-config.yml

```

在创建新的流水线时，提供流水线定义和包含登录名和密码的配置文件。在示例 15-7 中，给出了 .pipeline-config.yml 文件的内容。

示例15-7 Account微服务的.pipeline-config.yml配置文件

```

# The Cloud Foundry credentials where the microservice will be deployed
cf-url: https://api.run.pivotal.io
cf-username: replace
cf-password: replace
cf-org: replace

```

cf-space: replace

The Maven repository where versioned artifacts will be published

artifactory-url: https://cloudnativejava.jfrog.io/cloudnativejava/libs-release-local/

artifactory-username: replace

artifactory-password: replace

The git repository containing the microservice source code

git-source-repository-url: https://github.com/cloud-native-java/continuous-delivery

The git repository containing the version file

git-version-repository-url: git@github.com:cloud-native-java/continuous-delivery.git

git-version-private-key: |

-----BEGIN RSA PRIVATE KEY-----

REPLACE

-----END RSA PRIVATE KEY-----

这是一个包含 Account 微服务流水线配置参数的配置文件。在运行流水线之前，需要更新这些凭证，否则将无法完成流水线。前三个凭证不言自明，但对 `git-version-private-key` 需要解释一下。`git-version-private-key` 被提供给流水线，以便部署任务能够在流水线每次运行时 bump 发行版本。

图 15-5 显示了示例仓库的 `version` 分支，其中包含两个版本文件：一个用于 Account 微服务；另一个用于 User 微服务。

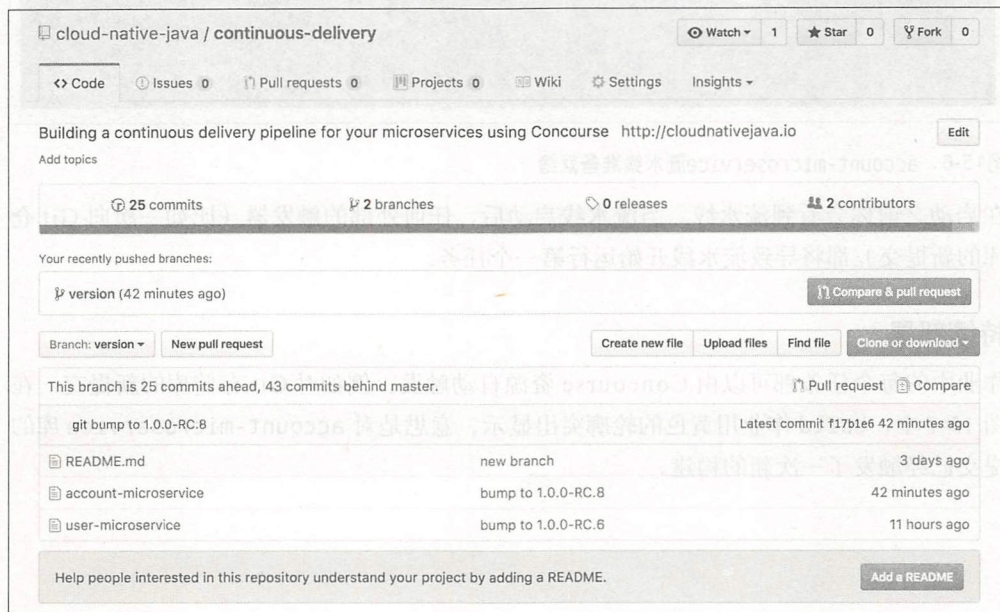


图15-5 示例GitHub存储库的version分支

现在，每个流水线将独立运行，每次流水线通过测试阶段时，相应的文件就会从当前版本“跳”到新版本。这个版本控制方法会有一个事件日志，我们可以使用它来自动回滚部署。这就是为什么需要我们提供 `git-version-private-key` 的原因，这样流水线中的任务可以在每次流水线运行时递增微服务的版本。

现在我们在 Concourse admin Web 界面中启动流水线。在你的浏览器中导航到 Concourse 实例的 Web 界面（根据你的安装情况而定）。

启动流水线

导航到你的 Concourse Web 界面后，将看到一个菜单，其中包含你创建的流水线列表。在创建 `account-microservice` 流水线之后，你会发现它已经准备好开始运行了——蓝色突出显示（参见图 15-6）。

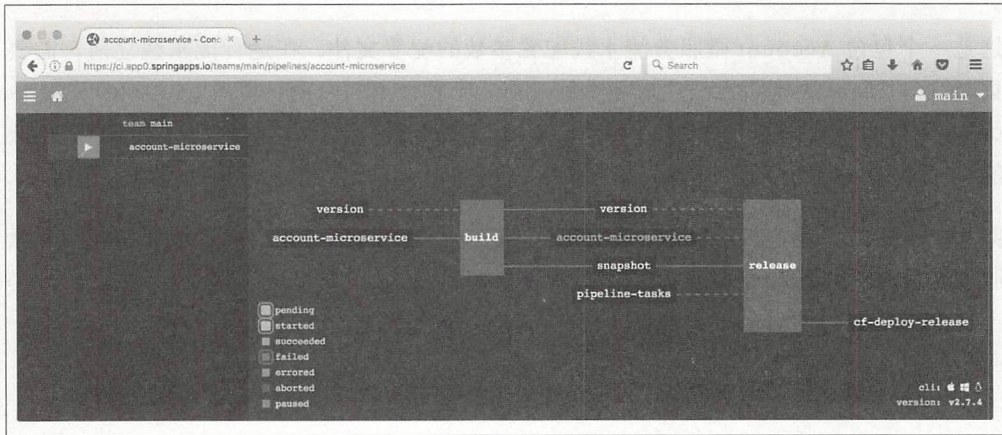


图15-6 `account-microservice`流水线准备就绪

在启动之前你会看到流水线。当流水线启动后，任何外部的触发器（比如一次向 Git 仓库的新提交）都将导致流水线开始运行第一个任务。

持续部署

作业中的每个任务都可以由 Concourse 资源自动触发，例如对 Git 存储库的新提交。在图 15-7 中，`build` 作业用黄色的轮廓突出显示，意思是对 `account-microservice` 库的提交已经触发了一次新的构建。

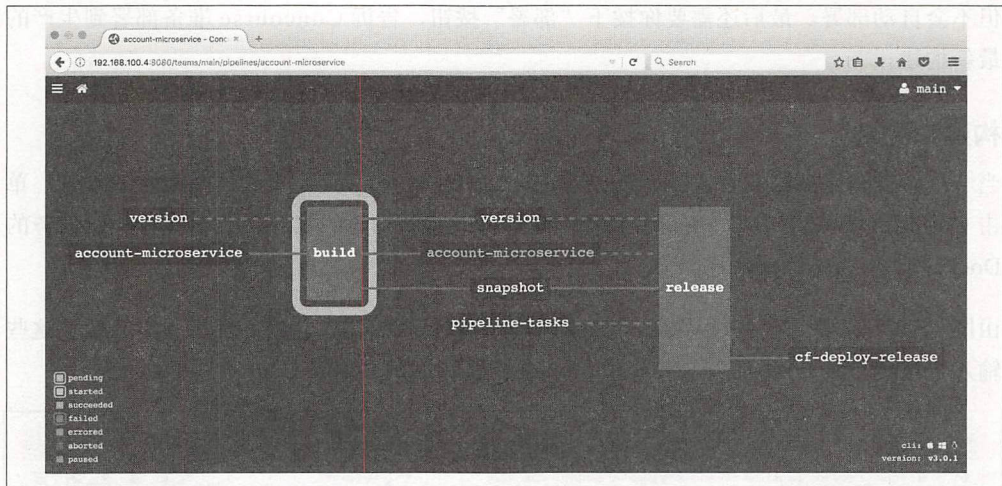


图15-7 account-microservice流水线在新的Git提交后启动

我们再来看一下 Account 微服务的 Concourse 流水线定义，看看它是如何工作的（示例 15-8）。

示例15-8 account-microservice流水线定义片段

```
resources:
- name: account-microservice ❶
  type: git
  source:
    uri: https://github.com/cloud-native-java/continuous-delivery
    branch: master
jobs:
- name: build
  plan:
  - get: account-microservice ❷
    trigger: true ❸
  - task: unit
    file: account-microservice/concourse/unit/unit.yml
```

- ❶ 描述将触发新构建的 Git 资源。
- ❷ 从 Git 中拉取主分支上的最新提交。
- ❸ 将其设置为 true 意味着任何新的提交都会触发此任务。

这是 pipeline.yml 文件的一段代码，描述了 Account 微服务的 Concourse 流水线。这段代码告诉 Concourse 如何从一个到 Git 仓库的新的提交自动触发一个作业的任务。

这是持续部署流水线的一个例子，这意味着流水线将在每次提交后自动构建、测试和部署新的代码。这与持续交付不同，其在每次新提交之后，只会为你准备一个新的构建，

但不会自动部署。最后还需要你按下“部署”按钮，告诉 Concourse 准备部署到生产的最新构建。

构建和测试

当一个新的构建被触发时，unit 任务将接收 pipeline.yml 定义文件中指定的输入。单击 Concourse Web 界面中的 build 作业，可以看到构建的状态，包括执行构建任务的 Docker 容器中的 stdout 流。

由图 15-8 可以看到，构建从 Account 微服务的源代码的输入和当前发行版本开始。这些输入将被传递到 unit 任务，并使用 Maven 构建和测试 Spring Boot 项目的源代码。

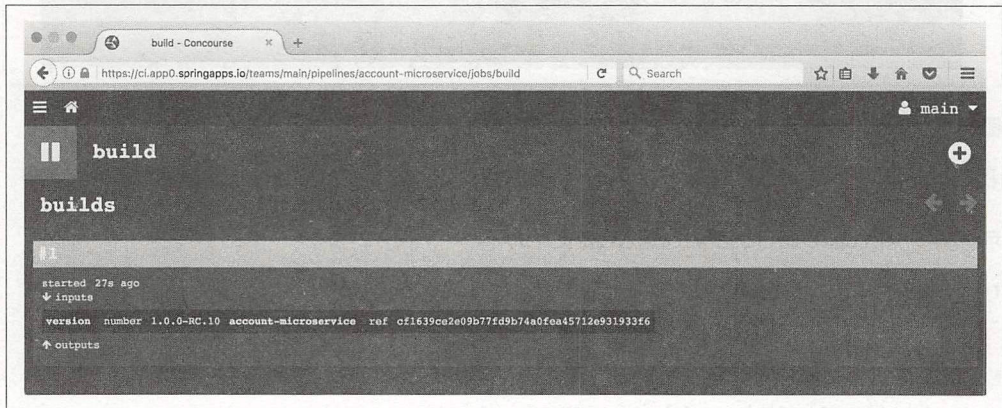


图15-8 第一个构建使用源代码输入触发

在图 15-9 中，当 Maven 构建过程开始时，可以看到 Docker 容器的输出流，Account 微服务正在下载 User 微服务的依赖。

版本化的 Maven 工件

在单元测试运行并通过后，生成的内部版本可以被自动准备为版本化工件。正如我们前面所讨论的，该流水线使用 Maven 工件库来存储将被部署到 Cloud Foundry 的版本化工件。在图 15-10 中，可以看到构建作业最终的成功状态。

如果构建失败，则会通知你，并且该作业将以红色突出显示。在我们的示例中，构建和测试是成功的，并且发布工件以递增的版本被上传到工件库中。

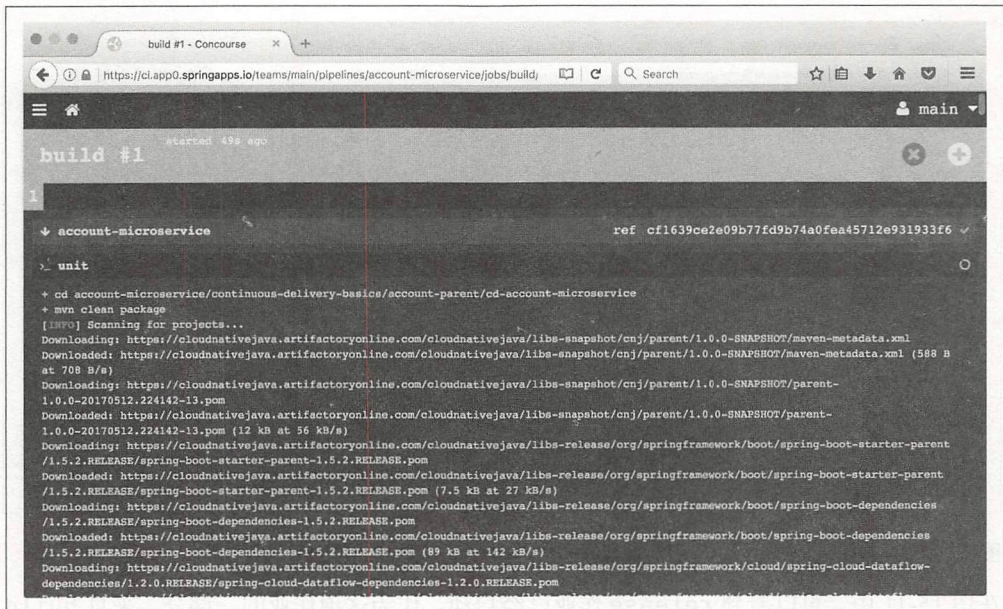


图15-9 unit任务构建和单元测试Spring Boot项目

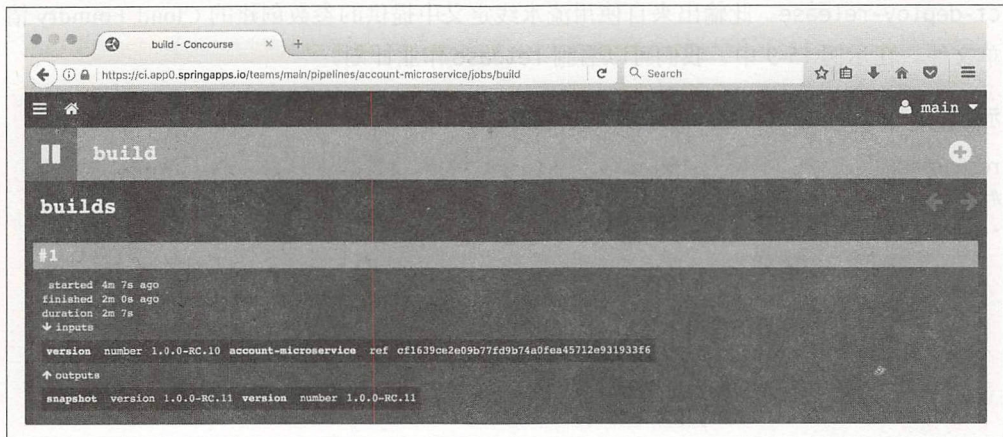


图15-10 构建完成并输出版本化的部署工件

部署到 Cloud Foundry

account-microservice 流水线的最后一步是准备部署到 Cloud Foundry 的发行版。在上一个作业中，我们构建并上传了 Spring Boot 应用程序的版本化部署工件。release 任务将由 Maven 工件库中的新版本触发，并启动部署到 Cloud Foundry（如图 15-11 所示）。

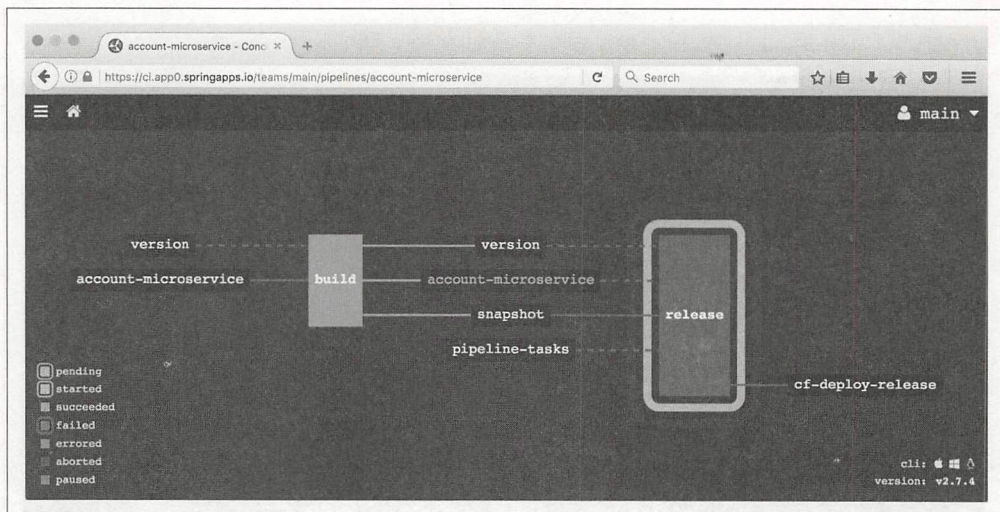


图15-11 构建任务之后，新版本被部署到Cloud Foundry

在图 15-11 中，可以看到 release 作业已经启动。作为这项作业的一部分，来自 build 作业的新发布的版本化工件将被发布到 Cloud Foundry 中。release 作业的最终输出是 cf-deploy-release。此输出来自使用流水线定义中提供的参数创建的 Cloud Foundry 清单文件。在示例 15-9 中，我们可以看到 release 作业计划。

示例15-9 account-microservice流水线中定义的发布作业

```
resources:
# ...
- name: snapshot
  type: maven-resource
  source:
    url: {{artifactory-url}}
    artifact: cnj:cd-account-microservice:jar
    username: {{artifactory-username}}
    password: {{artifactory-password}}
- name: cf-deploy-release
  type: cf # the Concourse resource for deploying to Cloud Foundry
  source:
    api: # \{{cf-url}}
    username: # \{{cf-username}}
    password: # \{{cf-password}}
    organization: # \{{cf-org}}
    space: # \{{cf-space}}
    skip_cert_check: false
# ...
jobs:
- name: release
```

```

serial: true
plan:
- get: snapshot
  trigger: true ❶
  passed: [build] ❷
- get: account-microservice
  passed: [build]
- get: version ❸
  passed: [build]
- get: pipeline-tasks # Resource used to create a custom manifest.yml
- task: prepare-manifest ❹
  file: account-microservice/concourse/release/prepare.yml
  params:
    MF_PATH: ../release-output/cd-account-microservice.jar
    MF_BUILDPACK: java_buildpack
- task: prepare-release
  file: account-microservice/concourse/release/release.yml ❺
- put: cf-deploy-release
  params:
    manifest: task-output/manifest.yml ❻

```

- ❶ 工件存储库中的新版本会触发发布。
- ❷ 该发布只能在构建任务通过后开始。
- ❸ 获取 build 作业输出的新版本。
- ❹ 准备一个自定义的 Cloud Foundry manifest.yml 部署文件。
- ❺ 将版本化的工件移动到暂存发布目录中。
- ❻ 将 manifest.yml 中描述的版本化工件部署到 Cloud Foundry 中。

这是一个 account-microservice 流水线的 pipeline.yml 定义中描述 release 作业的片段。release 任务正在编排一系列复杂的步骤，要求每个输入作为 build 作业的输出。下一步是创建一个自定义的 manifest.yml 文件，CF Concourse 资源需要这个文件来描述一个新的 Cloud Foundry 部署。名为 cf-deploy-release 的 CF 资源将作为 release 作业的输出，并将新版本的 account-microservice 部署到生产环境中。

在图 15-12 中，你可以看到一个成功发布的 account-microservice 版本示例。

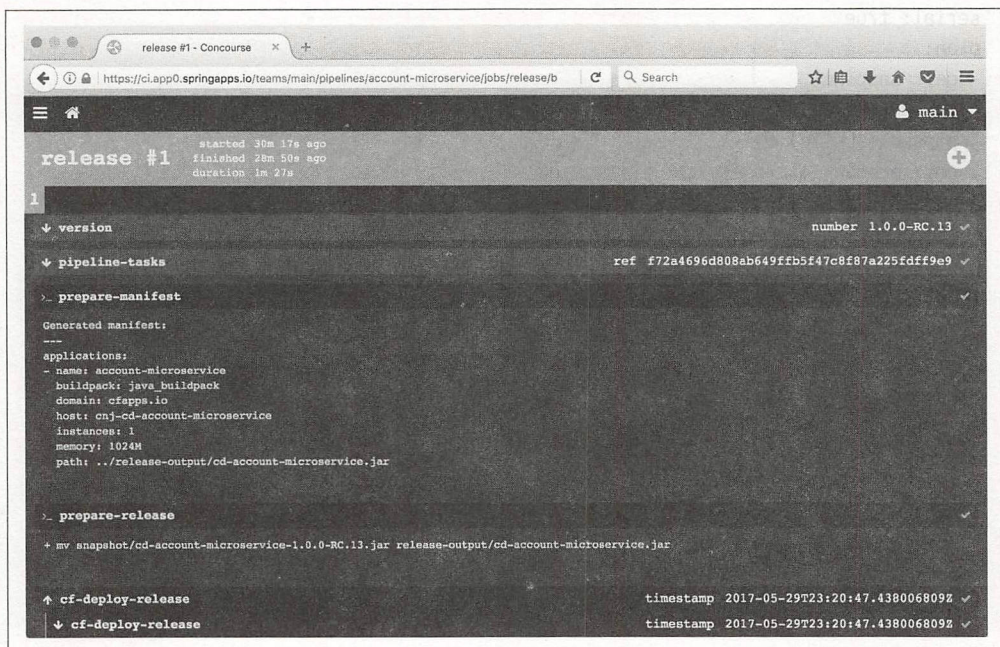


图15-12 release任务成功准备了一个新的Cloud Foundry部署



此次发布不是零宕机时间部署。你可以通过升级新的版本化工件来自定义 release 作业，以执行零宕机时间的蓝绿部署，然后将应用的 Cloud Foundry 路由从生产转变为灰度环境。

持续集成

前面我们探讨了如何为 Account 微服务设置基本的部署流水线。但是这个服务没有任何集成测试。可以使用 Concourse 建立复杂的集成环境，使用后台服务（如数据库）进行端到端测试或基本集成测试。下一节，我们将为 user-service 建立一个新的流水线，account-microservice 将依赖这个流水线。在这个新的流水线中，我们将建立一个额外的 integration 作业，为我们的两个微服务执行消费者驱动的协约测试。

消费者驱动的协约测试

如果没有相互制衡策略来确保团队不会连续发布重大变更，持续微服务交付会很快变得丑陋不堪。在第 4 章中，我们介绍了一个以消费者为中心的协约测试的例子。现在我们将对为 Account 微服务构建的简单 Concourse 流水线进行迭代，包括消费者驱动的测试步骤，以确保变更不会中断消费者。这个新流水线的目标是不断地将 user 服务与由

account 服务发布的由消费者驱动的协约测试集成。

为此，我们需要创建一个代码库，其中 *account* 服务可以为 *user* 服务发布消费者驱动的协约存根。正如我们在第 4 章中所讨论的，消费者驱动的协约测试是由服务依赖的消费者创建的。我们每个消费者都将负责将消费者驱动的协约测试发布到指定的存储库。我们来看一个例子。

我们是 *account* 服务的所有者，我们依赖 *user* 服务。作为消费者，我们要确保 *user* 服务不会发布导致我们的服务失败的任何重大变更。为了实现集成测试的自动化，我们将把消费者测试发布到由 *user* 服务拥有的指定的存储库中。因此，作为消费者，我们有责任将我们的消费驱动测试发布到我们依赖的服务的集成存储库中。否则，*user* 服务将不知道我们对 *account* 服务的期望。

User 微服务流水线

看看我们之前为 *account* 服务创建的流水线设计，修改它以包含 *user* 服务的消费者驱动协约测试。在图 15-13 中，可以看到 User 微服务的流水线。

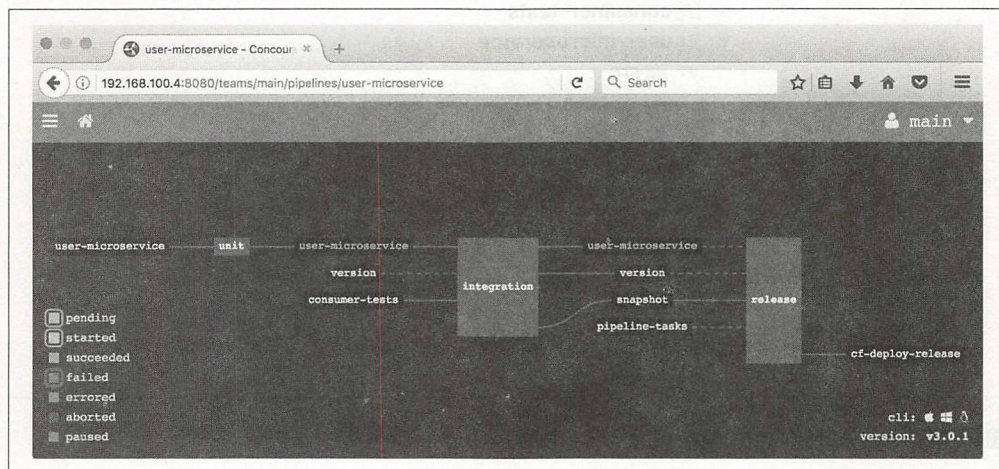


图15-13 User微服务的Concourse流水线

在这个服务中，有一个额外的流水线步骤：*integration* 作业。在集成作业中，在打包新版本工件并将其上传到 Maven 工件库之前，将针对 User 微服务执行一组消费者驱动测试。

在图 15-14 中，可以看到 User 微服务的源代码目录。

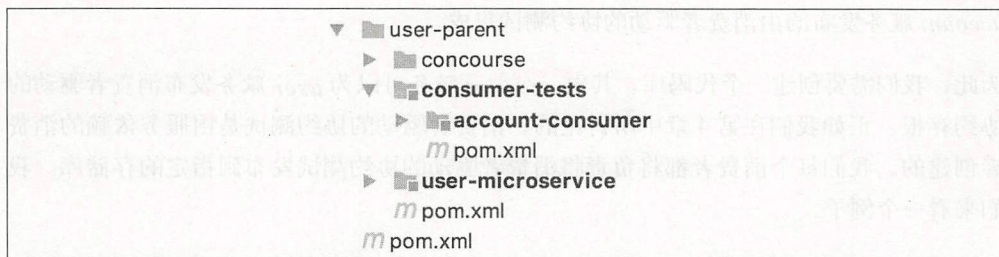


图15-14 User微服务的源代码目录

对于 User 微服务，我们将有一个额外的目录，其中包含消费者发布的由消费者驱动的协约测试集合。在此之前，我们需要在 User 微服务中发布协议存根，以便其他服务可以针对其编写测试。

在图 15-15 中，可以看到使用 Spring Cloud Contract 的消费者驱动的协约存根，我们在第 4 章中提到过。

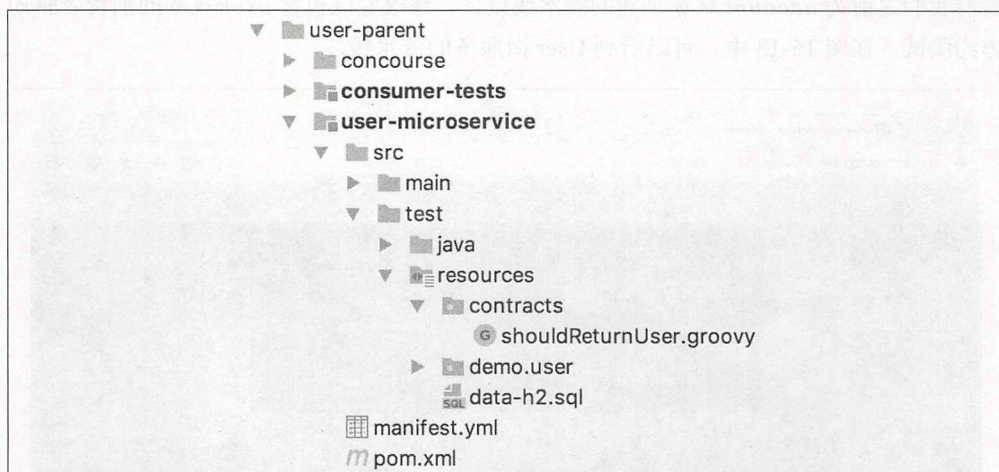


图15-15 User微服务发布的消费者驱动的协约存根

当 User 微服务被发布时，可以使用该存根来运行模拟 User 微服务行为的嵌入式应用程序服务器。

在图 15-16 中，你可以看到 Account 微服务编写的消费者驱动的协约测试。User 微服务流水线中的 integration 任务将运行此目录中的所有消费者测试，以确保任何新的变更不会中断消费者。编写这些测试是每个消费者的责任，如果测试未通过，则 User 微服务将无法发布对生产的重大变更。

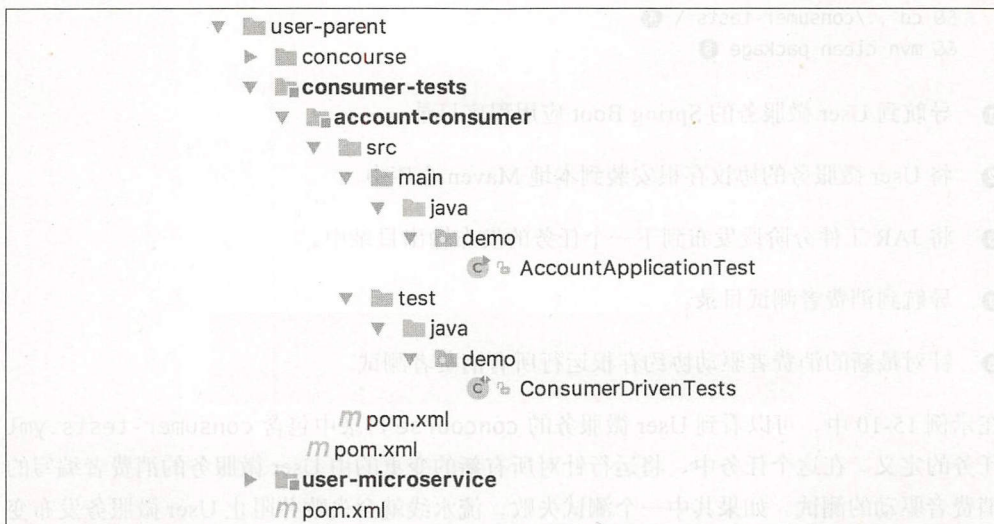


图15-16 Account微服务针对User微服务编写消费者测试

现在来看看 `consumer-tests` 任务（见示例 15-10），它是 User 微服务 Concourse 流水线中 `integration` 作业的一部分。

示例15-10 User微服务中的集成任务

This task will run the integration tests and consumer-driven contract tests of
the user microservice and its consumers.

platform: linux

image_resource:

type: docker-image

source:

repository: maven

tag: alpine

inputs:

- name: consumer-tests

outputs:

- name: release

run:

path: sh

args:

- -exc

- |

cd consumer-tests/user-microservice \ ①

&& mvn clean install -DskipTests \ ②

&& mv target/user-microservice.jar \

../../release/user-microservice.jar \ ③


```
&& cd ../consumer-tests \ ④  
&& mvn clean package ⑤
```

- ① 导航到 User 微服务的 Spring Boot 应用程序目录。
- ② 将 User 微服务的协议存根安装到本地 Maven 仓库中。
- ③ 将 JAR 工件分阶段发布到下一个任务的发布输出目录中。
- ④ 导航到消费者测试目录。
- ⑤ 针对最新的消费者驱动协议存根运行所有消费者测试。

在示例 15-10 中，可以看到 User 微服务的 `concourse` 目录中包含 `consumer-tests.yml` 任务的定义。在这个任务中，将运行针对所有新的变更的由 User 微服务的消费者编写的消费者驱动的测试。如果其中一个测试失败，流水线就会失败并阻止 User 微服务发布变更。通过使用持续交付流水线来自动执行这些类型的集成测试，微服务所有者必须在消费者应用程序被部署到生产之前，主动解决与消费者有关的重大变更。



为使示例代码片段易于阅读，本章中使用的示例中的文件夹路径可能与本书其他代码示例不同。任务的输入资源将使用微服务的完整路径。在本书中，我们使用单个存储库是为了便于讲解。对于你的生产微服务，不建议在同一个 Git 存储库中包含多个微服务。

数据

我们还没有接触蓝绿部署中的数据库迁移问题。在此我们只能讲一些皮毛，因为没有万能解决方案。

在应用了数据库模式突变的数据库中，如 Flyway(<https://flywaydb.org>)和 Liquibase(<http://www.liquibase.org>) 这样的 RDBMS 模式 (schema) 迁移工具会为其维护一个清单，如果确定有新版本可用，则将数据库模式更新为新版本。如果只是向数据库中添加新的列、表或记录，那么需要回滚时也没有什么可担心的。你仍然拥有以前的所有记录和模式。

如果在部署新版本的服务 v2 时，使用了支持版本模式变化和将现有数据库从一个版本迁移到另一个版本的工具，如 Flyway 和 Liquibase 这样的工具，将数据库模式演化为版本 2.0 时会发生什么？如果出现错误，并且对模式进行了破坏性更新（例如更改列或删除记录），又会发生什么情况？

设计蓝绿部署时需要注意这些问题。不要删除数据，而应考虑使用墓碑 (tombstone) 记录，其中应用程序只是逻辑删除记录（可能将 DELETED 列设置为 TRUE），但保留数据，以便

应用程序可以通过取消墓碑列标记来重新生成记录。如果你要改变一个列记录，你可以将旧数据复制到一个新列的同时保持这两列吗？

尽一切可能使代码可以回滚到更早的版本，因为你永远不知道什么时候会失败。但实际上，你不需要支持回滚到以前的所有版本，只需要回滚到最后一个版本即可。团队经常给自己足够的回退空间，以便回滚到最近一个已知模式的版本 1。在版本 2 中，弃用的列和记录被保留，驻留在那里。在版本 3 中，可以应用在版本 1 中不应该发生的破坏，从版本 1 中永久移植。

生产

“完成”的软件就像是“割过草”的草坪。

—— Jim Benson (<https://twitter.com/ourfounder/status/770075137332932608>)

在本书中，我们着眼于构建云原生应用程序。云原生应用程序是应用云原生模式，注定要在云平台上运行的应用程序。它是经过设计、优化并易于迭代和可以快速反馈的应用程序。它从来没有完成状态。云原生应用程序(希望)从概念到生产将会经历许多次更新，每一次更新都会为客户带来更多的价值。在本章中，我们研究了如何通过持续交付尽可能地降低更新成本。

第 V 部分

附录

在Java EE中使用Spring Boot

在本附录中，我们将介绍如何将 Spring Boot 应用程序与 Java EE 集成。Java EE 是一组 API 的伞式名称，有时是运行时的 Java EE 应用程序服务器。像红帽的 WildFly AS(<http://wildfly.org>) 这样的 Java EE 应用服务器（以前称为 JBoss Application Server 的应用服务器）提供了这些 API 的实现。我们将研究如何构建利用 Java EE 应用程序服务器之外的 Java EE API 的应用程序。如果当前你正在构建一个全新的应用程序，则不需要阅读此附录。本附录对于那些深陷于应用服务器中，想要迁移到微服务架构的用户更为有用。有关将遗留应用程序迁移到 Cloud Foundry 等云平台的讨论，请参见第 5 章中的内容。

在实践中，Spring 充当 Java EE API 的消费者。它不请求任何一个 Java EE API。只要有可能，Spring 都支持独立于完整的 Java EE 应用程序服务器之外来单独使用 Java EE API。在理想情况下，Spring 应用程序可以跨环境进行移植，包括嵌入式 Web 应用程序、应用程序服务器以及几乎所有平台即服务（PaaS）产品。

兼容性和稳定性

Spring 4.2（Spring Boot 1.3 及更高的基准版本）支持 Java SE 6 或更高版本（具体来说，最低 API 级别是 2010 年初发布的 JDK 6u18）。对于 Java SE 6 和 Java SE 7，Oracle 已经停止发布更新（除了安全修复之外）。请考虑转移到 Java SE 8。

Spring 还支持 Java EE 6+（2009 年推出）。实际上，Spring 4 着眼于 Servlet 3.0+，但也兼容 Servlet 2.5。基于 Spring 构建的 Spring Boot 最适合于 Servlet 3.1 或更高版本。Spring Boot 的依赖方式可以确保你能尽快获得最新、最好的服务，不过你仍然可以恢复到旧的 Servlet 3.0 代服务器。

市场一直青睐于 Apache Tomcat (<http://tomcat.apache.org>)（Pivotal 贡献巨大的项目）、Eclipse 的 Jetty 和 Red Hat 的 Wildfly。大多数 Java 开发人员都不会部署完整兼容 Java

EE 的容器,因此 Spring 引入了大量不运行 Java EE 应用程序服务器的 Java EE API。显然,Java EE 提供了一些非常引人注目的 API (根深蒂固的)。仅举几例,Servlet API、JSR 330 (javax.inject) 和 JSR 303 (javax.validation)、JCache、JDBC、JPA、JMS、JAX-RS 等非常有用,对用户来说也很实用, Spring 和 Spring Boot 都支持它们。

随着开发人员转向云,并因此转向云原生架构,我们看到依靠 Java EE 应用程序服务器的价值开始下降。全盘接受的方法违背了由小型、单一聚焦、可扩展的服务组成的云原生系统构建原则。

很多 Java EE API 非常有用,但是它们的进化速度非常缓慢。这表明进化缓慢且标准化的 Java EE API 最好是用在几乎没有任何波动性的层面上。JDBC 为几十年前的基于 SQL 的数据库提供了中间件绑定,这是一个完美合理的 API 的例子。为 HTTP 应用程序提供中间件绑定的 Servlet API 也是一个功能非常强大的 API。JDBC 和 Servlet 规范都是 API 的示例,不需要经常更改。

除此之外,人们也在拼命努力。新的开发模式在不断涌现。截至 2017 年年中,HTTP 2 (几乎可以免费提供性能改进) 已经在其他很多平台和技术上支持了很多年,但是在 Servlet 规范中还没有提供支持。开发人员必须在底层的 Servlet 容器中使用本地 API 来获取这个功能 (暂时)。随着应用程序要面对不断增长的需求和更大的数据集,当工作可以被反应式地处理时,人们已经不再能接受阻塞等待输入和输出的线程,等待释放出宝贵的资源。要使反应式编程工作,请求处理链中的每个链接都必须支持反应性——从基于 HTTP Servlet 的请求一直到 JDBC 数据库。目前,也没有对此的支持。Oracle 宣称在未来的 JDBC 版本中可以看到对反应式编程的支持,Java EE 8 将支持 HTTP 2。所以,最终,我们也会实现。

软件是为目的服务的,通常是商业目的,而商业是不会依靠有几十年历史的劣质解决方案来竞争。拥抱标准,只有这样才有意义。

下面我们来看几个常用的 API,你会发现它们在 Spring Boot 应用程序中表现得非常出色。

JSR 330 (和 JSR 250) 的依赖注入

Spring 一直使用各种方法来提供配置。Spring 根本不在乎它在哪里了解你的对象,以及如何将它们连接在一起。一开始,有 XML 配置格式。后来, Spring 引入了组件扫描来发现和注册具有构造型注释的组件,比如 @Component, 或者任何其他用 @Component 注解的注解,比如 @RestController 或 @Service。2006 年, Spring Java Configuration 项目诞生了。这种方法将 bean 带入 Spring, 使用从 @Bean 注解中注解的方法返回的对象。你可以调用这些 bean 定义提供者方法。

与此同时, Google 的 Bob Lee 引入了 Guice, 它也支持 bean 定义提供者方法的概念, 类

似于 Spring Java 配置项目。这两个项目以及其他一些项目是独立演变的，每个项目都有庞大的社区。

时间来到 2007 年、2008 年和 2009 年，Java EE 6 在这段时间迅速成长。开发了自己的依赖注入技术的 JBoss 团队的 Seam 试图定义一个依赖注入技术标准——JSR 299 (CDI)。Seam 提出的 JSR 299 跟 Spring 和 Guice 这两个最受欢迎的技术不同，所以 Spring 的创始人 Rod Johnson 和 Guice 的创始人 Bob Lee 提出了 JSR 330。JSR 330 定义了一套通用的影响业务组件代码的依赖注入注解框架。这让每个依赖注入容器来区分引擎本身的实现。

Spring 和 Guice 原生支持 JSR 330，最终也支持 JSR 299 实现。事实上，其他的依赖注入技术如 Dagger(它为编译时代码生成而优化)以及像 Android 这样的移动环境也支持它。如果你绝对必须有可移植的依赖注入，请使用 JSR 330。

假设在类路径中有 `javax.inject:javax-inject:1` 定义，并解析和注入 bean 引用，那么通常会在 JSR 330 中使用一些注解。

- `@Inject` 等价于 Spring 的 `@Autowired` 注解。它识别可注入的构造函数、方法和字段。
- `@Named` 等同于 Spring 的各种构造型注解，如 `@Component`。它标记一个要在容器中注册的 bean，它可以给这个 bean 提供一个用来注册的基于字符串的 ID。
- `@Qualifier` 可以用来按类型（或字符串 ID）限定 bean。自然，这与 Spring 的 `@Qualifier` 注解基本一样。
- `@Scope` 类似于 Spring 的 `@Scope` 注解，用于告诉容器 bean 的生命周期是什么。例如，你可以指定一个 bean 是 `session` 的作用域，也就是说，它应该在 HTTP 请求的线上生存和死亡。
- `@Singleton` 告诉容器该 bean 只应该被实例化一次。这是 Spring 的默认设置，但这是一个很普遍的概念，值得所有的实现都支持。

JSR 330 还定义了一个 `javax.inject.Provider <T>` 接口。业务组件可以直接注入一个给定 bean 的实例，也可以使用 `Provider <Foo>` 来注入。这类似于 Spring 的 `ObjectFactory <T>` 类型。与直接注入实例相比，`Provider <T>` 实例可以用来检索给定 bean 的多个实例，处理 lazy 语义，打破循环依赖和包含范围的抽象。



JSR 250 来自 Java EE 5，如果该类型位于类路径上（在 JDK 的新版本中），它也受到原生支持。例如，如果你使用过 `@javax.annotation.Resource`，则可能看到过这些注解。这些注解通常在 EJB 3 代码中使用，但笔者从来没有真正看到在其他地方使用它们。这可以让开发人员轻松地 EJB 3 环境中迁移代码，在 EJB 3 环境中引用解析是通过 `@Resource` 来发送的。

在 Spring Boot 应用程序中使用 Servlet API

在默认情况下，当你引入 `spring-boot-starter-web` 时，Spring Boot 会自动为你配置一个 Web 运行时环境。这个运行时环境在 5.0 之前的 Spring 版本中是一个 Servlet 容器，比如 Apache Tomcat、Red Hat 的 Undertow 和 Eclipse 的 Jetty。



Spring 框架 5.0 及更高版本还支持在 Netty 上构建的反应式运行时环境。但这不是本书的话题。

这些都是很好的选择。Apache Tomcat 的使用最普遍，但选择其他实现也是可以的。Spring Boot 支持常见的应用，比如使用 `server.port` 属性指定端口，使用 `server.context-path` 指定上下文路径，使用 `server.ssl.***` 配置 SSL 支持，使用 `server.compression.***` 启用 GZip 压缩。特别是 GZip 压缩需要像 JSON 那样的多种有效负载传送 HTTP 并压缩它们。浏览器知道如何解压缩这些资源。这为大型或静态有效载荷（如 JavaScript 或 CSS 文件）带来了高效率。如果你想压缩所有的静态资源（Spring Boot 应用在 `src/main/resources/***` 目录下），你只需要指定 `spring.resources.chain.gziped=true` 即可。如果要压缩动态端点（如 JSON 负载），则可以使用 `server.compression.***` 属性。示例 A-1 显示如何压缩所有 JSON 结果。

示例A-1 GZip压缩所有JSON负载

```
server.compression.enabled=true
server.compression.mime-types=application/json
```

你访问端点而不指定 `accept-encoding` 和 `accept-encoding` 就可以做到这样。我们的示例应用程序在 `/mvc/hi` 下启用了 Spring MVC 端点，它会返回 JSON 输出。调用那个没有指定 `accept-encoding` 头的端点，你会得到原始的 JSON。增加 `accept-encoding` header 调用它并指定 `gzip`，你将得到二进制（压缩）的数据，必须把该数据传递给 `gunzip` 来读取（见示例 A-2）。

示例A-2 使用和不使用accept-encoding header来调用REST端点

```
> curl http://localhost:8080/mvc/hi
{"greetings":"Hello, world!"}

> curl -H"accept-encoding: gzip" http://localhost:8080/mvc/hi
?VJ/JM-??K/V?R?H???Q(?/?IQT???X

> curl -H"accept-encoding: gzip" http://localhost:8080/mvc/hi | gunzip
{"greetings":"Hello, world!"}
```

容器还有它特有的属性。从一个容器到另一个容器，这些属性必然是不一致的。如果你部署到 Undertow，则可以指定 `server.undertow.worker-threads=10` 来限制创建的工作线程数量。如果你部署到 Tomcat，则可以使用 `server.tomcat.max-threads=10` 来限制正在运行的线程数。如果你部署到 Eclipse Jetty 项目，那么可以使用 `server.jetty.acceptors` 和 `server.jetty.selectors` 来限制 Jetty 维护的接受者和选择器线程的数量。

公共属性和这些容器特有的属性应该已经能够表达大部分你以前可能拥有的配置。如果不能满足需求，并且想要以编程的方式定制或询问嵌入式 servlet 容器引擎，那么可以考虑实现一个 `EmbeddedServletContainerCustomizer` 接口，这是一个回调接口，可以让你访问 Spring Boot 配置的实现。Spring Boot 也支持容器特定的回调接口。这些特定于容器的回调是一个覆盖或创建应用程序本身开发中涉及的组件的时机。

在示例 A-3 中，我们将解引用容器特定的定制器回调并对其进行迭代。你可以将这些定制器添加到这些容器实例的集合中来创建自己的定制器。应该为要访问的定制器提供一个 `container#add*` 方法。

示例A-3 使用EmbeddedServletContainerCustomizer中的嵌入式容器

```
package servlets;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
//@formatter:off
import org.springframework.boot.context.embedded.AbstractEmbeddedServletContainerFactory;
import org.springframework.boot.context.embedded.ConfigurableEmbeddedServletContainer;
import org.springframework.boot.context.embedded
    .EmbeddedServletContainerCustomizer;
import org.springframework.boot.context.embedded
    .jetty.JettyEmbeddedServletContainerFactory;
import org.springframework.boot.context.embedded
    .tomcat.TomcatEmbeddedServletContainerFactory;
import org.springframework.boot.context.embedded.undertow.UndertowEmbeddedServletContainerFactory;
//@formatter:on
import org.springframework.stereotype.Component;

@Component
class ContainerAnalyzer implements EmbeddedServletContainerCustomizer {

    private final Log log = LogFactory.getLog(getClass());

    @Override
    public void customize(ConfigurableEmbeddedServletContainer c) {
```



```
this.log.info("inside " + getClass().getName());
```

❶

```
AbstractEmbeddedServletContainerFactory base = AbstractEmbeddedServletContainerFactory.class
```

```
.cast(c);
```

```
this.log.info("the container's running on port " + base.getPort());
```

```
this.log.info("the container's context-path is " + base.getContextPath());
```

❷

```
if (UndertowEmbeddedServletContainerFactory.class.isAssignableFrom(c.getClass())) {
```

```
UndertowEmbeddedServletContainerFactory undertow = UndertowEmbeddedServletContainerFactory.class
```

```
.cast(c);
```

```
undertow.getDeploymentInfoCustomizers().forEach(
```

```
dic -> log.info("undertow deployment info customizer " + dic));
```

```
undertow.getBuilderCustomizers().forEach(
```

```
bc -> log.info("undertow builder customizer " + bc));
```

```
}
```

❸

```
if (TomcatEmbeddedServletContainerFactory.class
```

```
.isAssignableFrom(c.getClass())) {
```

```
TomcatEmbeddedServletContainerFactory tomcat = TomcatEmbeddedServletContainerFactory.class
```

```
.cast(c);
```

```
tomcat.getTomcatConnectorCustomizers().forEach(
```

```
cc -> log.info("tomcat connector customizer " + cc));
```

```
tomcat.getTomcatContextCustomizers().forEach(
```

```
cc -> log.info("tomcat context customizer " + cc));
```

```
}
```

❹

```
if (JettyEmbeddedServletContainerFactory.class.isAssignableFrom(c.getClass())) {
```

```
JettyEmbeddedServletContainerFactory jetty = JettyEmbeddedServletContainerFactory.class
```

```
.cast(c);
```

```
jetty.getServerCustomizers().forEach(
```

```
cc -> log.info("jetty server customizer " + cc));
```

```
}
```

```
}
```

```
}
```

❶ 将容器向下转换为抽象基类型是非常有用的，这样我们不止可以调用各种 `set*` 方法。

② Undertow 的特定容器实现

③ Apache Tomcat

④ Eclipse Jetty

Spring Boot 自动创建嵌入的 servlet 容器，然后使用 Servlet 3.0 程序注册器注册相关的 Spring 堆栈设备。Spring Boot 将注册 Spring MVC DispatcherServlet，无论 Spring Security 需要什么过滤器，以及 Spring 生态系统其他部分都需要什么其他机制。这一切都会为你自动完成。你不需要配置 web.xml 或提供管理容器本身的配置。如果你使用的是 Spring 项目（比如 Spring MVC），那么就已经计量好了。你也可以创建自己的 servlet 或过滤器。Spring Boot 将智能地分别注册类型为 `javax.servlet.Filter` 或 `javax.servlet.Servlet` 的任何 Spring bean（由 `@Bean` 提供者方法贡献，或者使用构造型注释进行注释）作为过滤器和 servlet。

如果你想要控制 servlet 或过滤器的注册方式（它们的排序、URL 映射、参数等），那么有两个选择。你可以使用 `ServletRegistrationBean` 实例包装你的 servlet，让 Spring 帮你管理。你还可以使用 `FilterRegistrationBean` 实例来实现相同的过滤器。假设我们有一个 `javax.servlet.Filter`，名为 `LoggingFilter`（见示例 A-4）。

示例A-4 常见的日志记录例子

```
package servlets;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import javax.servlet.*;
import java.io.IOException;
import java.time.Instant;

class LoggingFilter implements Filter {

    private final Log log = LogFactory.getLog(getClass());

    @Override
    public void init(FilterConfig config) throws ServletException {
        this.log.info("init()");
        String initParameter = config.getInitParameter("instant-initialized");
        Instant initializationInstant = Instant.parse(initParameter);
        this.log.info(Instant.class.getName() + " initialized "
            + initializationInstant.toString());
    }
}
```



```

@Override
public void doFilter(ServletRequest req, ServletResponse resp,
    FilterChain chain) throws IOException, ServletException {
    this.log.info("before doFilter(" + req + ", " + resp + ")");
    chain.doFilter(req, resp);
    this.log.info("after doFilter(" + req + ", " + resp + ")");
}

@Override
public void destroy() {
    log.info("destroy()");
}
}

```

你可以使用 `FilterRegistrationBean` bean 定义注册实例，指定初始化参数、过滤器是否应该支持异步操作，以及过滤器应用于哪个 URL 路径等（见示例 A-5）。

示例A-5 显式注册过滤器和servlet

```

package servlets;

import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;

import java.time.Instant;

@Configuration
public class FilterConfiguration {

    @Bean
    FilterRegistrationBean filter() {
        LoggingFilter filter = new LoggingFilter(); ❶
        FilterRegistrationBean registrationBean = new FilterRegistrationBean(filter);
        registrationBean.setOrder(Ordered.HIGHEST_PRECEDENCE);
        ❷
        registrationBean.addInitParameter("instant-initialized", Instant.now()
            .toString());
        return registrationBean;
    }
}

```

- ❶ 在这里实例化一个 `Filter` 的实例。你可以像 Spring bean 一样管理 bean 的生命周期。
- ❷ 我们在这里编程指定初始化参数。这些参数可能来自在命令行上指定的配置或来自 Spring Cloud Config Server。



Spring 会确保在适当的时候调用 `init` 和 `destroy` 方法。

Spring Boot 甚至支持自动检测和管理使用 Servlet 3.0 注释的 `@javax.servlet.annotation.WebServlet`、`@javax.servlet.annotation.WebFilter` 和 `@javax.servlet.annotation.WebListener` 注释的组件。将 `@org.springframework.boot.web.servlet.ServletComponentScan` 添加到 Spring 配置类来激活该支持，最好是和 `@SpringBootApplication` 所在的类一样。Spring 将管理发现的类，就好像类已经直接在 Spring 中注册了一样。你不需要使用另一个构造型注释（如 `@Component`）或者使用注册 bean（示例 A-6）注释这些组件。



很明显，不建议把新的代码写成 Servlet 实例，因为你可以使用 Spring MVC 实现同样的事情，如果你想让现有的代码工作，这可能会有帮助。

示例A-6 Spring为我们管理@WebServlet组件

```
package servlets;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

❶

```
@WebServlet(urlPatterns = "/servlets/hi", asyncSupported = false)
class GreetingsServlet extends HttpServlet {
```

```
    private final Log log = LogFactory.getLog(getClass());
```

```
    @Override
```

```
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        this.log.info("doGet(" + req + ", " + resp + ")");
        resp.setStatus(200);
        resp.setHeader("Content-Type", "application/json");
```



```

    resp.getWriter().println("{ \"greeting\" : \"Hello, world\"}");
    resp.getWriter().close();
}
}

```

- ① 你可以在 web.xml 中指定大部分的配置，或者在相应的 Servlet API 注解类型的注册 bean 中指定。

使用 JAX-RS 构建 REST API (Jersey)

Spring Boot 使得使用 JAX-RS 创建 REST API 变得非常简单。JAX-RS 是一个标准，并且需要一个实现。示例 A-7 演示了 GreetingEndpoint 中用于 Jersey 2.x (<http://bit.ly/2vmlUmk>) 的 Boot 的 JAX-RS 自动配置。这个例子使用了 Spring Boot starter org.springframework.boot:spring-boot-starter-jersey。如果你想使用其他的 JAX-RS 实现（如 REST Easy），那么在 Spring Boot 应用程序中安装注册过滤器和 servlet 也很简单。

示例A-7 JAX-RS GreetingEndpoint

```

package demo;

import javax.inject.Inject;
import javax.inject.Named;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

@Named
①
@Path("/hello")
②
@Produces({ MediaType.APPLICATION_JSON })
③
public class GreetingEndpoint {

    @Inject
    private GreetingService greetingService;

    @POST
    ④
    public void post(@QueryParam("name") String name) { ⑤
        this.greetingService.createGreeting(name);
    }

    @GET
    @Path("/{id}")
    public Greeting get(@PathParam("id") Long id) {

```

```

    return this.greetingService.find(id);
}
}

```

- ❶ JSR 330 的 `@Inject` 注解。
- ❷ JAX-RS 的 `@Path` 注解在功能上等同于 Spring MVC 的 `@RequestMapping` 注解。它告诉容器该端点应该暴露的路由。
- ❸ Spring MVC 的 `@RequestMapping` 注解提供了 `produces` 和 `consumes` 属性，可以让你指定给定端点可以使用或生成的内容类型。在 JAX-RS 中，这个映射是通过独立注解完成的。
- ❹ HTTP 动词，也是在 Spring MVC 的 `@RequestMapping` 注解中指定的，在这里被指定为一个独立的注解。
- ❺ `@QueryParam` 注解告诉 JAX-RS 注入任何传入的请求参数（`?name=.`）作为方法参数。在 Spring MVC 中，你可以使用 `@RequestParam` 注释的方法参数。

Jersey 需要一个 `ResourceConfig` 子类来启用基本特性和注册组件（见示例 A-8）。

示例A-8 配置Jersey的ResourceConfig子类

```

package demo;

import org.glassfish.jersey.jackson.JacksonFeature;
import org.glassfish.jersey.server.ResourceConfig;

import javax.inject.Named;

❶
@Named
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        this.register(GreetingEndpoint.class); ❷
        this.register(JacksonFeature.class); ❸
    }
}

```

- ❶ 我们使用 JSR 330 来注册端点，尽管在这里我们可以很容易地使用 Spring 的标注。它们可以互换。
- ❷ 需要明确注册我们的 JAX-RS 端点。

- ③ 需要告诉 JAX-RS 我们要处理 JSON 编码。Java EE 没有用于编组 JSON 的内置 API (正如我们上面所讨论的, 暂时在 Java EE 8 中提供它), 但是你可以使用 Jersey 特有的特性实现来插入流行的 JSON 编码 (例如 Jackson)。

Spring Boot 自动配置 Jersey 的 `org.glassfish.jersey.servlet.ServletContainer` 来监听相对于应用程序根目录的所有请求。在 JAX-RS 中, 消息编码和解码分别通过 `javax.ws.rs.ext.MessageBodyWriter` 和 `javax.ws.rs.ext.MessageBodyReader` SPI 来完成, 有点类似于 Spring MVC 的 `org.springframework.http.converter.HttpMessageConverter` 层次结构。在默认情况下, JAX-RS 没有启用许多有用的消息体读取器和编写器。在我们的例子中在 `ResourceConfig` 子类中注册一个 `JsonFeature` 来支持 JSON 编码和解码。

JTA 和 XA 事务管理

事务管理对于很多人来说是一个令人困惑的问题, 特别是有这么多的选择!

使用 Spring 的 PlatformTransactionManager 的本地资源事务

从根本上说, 事务处理大多以相同的方式工作: 客户端开始工作, 做一些工作, 提交工作。如果出现问题, 恢复 (回滚) 事务, 并将基础事务资源重置为开始工作之前的状态。不同资源的实现差异很大。JMS 客户端创建一个事务会话, 然后提交。JDBC Connection 可以被设置为非自动提交工作, 这与批处理工作是一样的, 然后在明确指示的情况下提交。

本地资源事务应该是事务管理的默认方法。你可以使用具有各种事务性 Java EE API (如 JMS、JPA、CCI 和 JDBC) 的本地资源事务。Spring 支持许多像 AMQPbrokers 的事务资源, 例如 RabbitMQ (<http://rabbitmq.org>)、Neo4j 图形数据库 (<http://neo4j.com/>) 和 Pivotal Gemfire (<http://www.pivotal.io/big-data/pivotal-gemfire>)。不幸的是, 这些事务资源中提供的用于启动、提交或回滚工作的 API 都不相同。为了简化, Spring 提供了 PlatformTransactionManager 层次结构。PlatformTransactionManager 有许多可插入的实现, 它们将事务的各种概念调整为一个通用的 API。Spring 能够通过这个层次结构的实现来管理事务。

Spring 为事务提供分层支持。在最底层, Spring 提供 TransactionTemplate。TransactionTemplate 封装了一个 PlatformTransactionManager bean, 并使用它来管理事务, 客户端提供一个工作单元作为 TransactionCallback 的一个实现。我们先把所有东西连接起来。有以下两个例子。数据库 DataSource、DataSourceInitializer、JdbcTemplate、PlatformTransactionManager 等, 这些都是通用的, 所以我们将它们

定义在一个共享的 Java 配置类中。这个 Java 配置还定义了一个 RowMapper，这是一个 JdbcTemplate 回调接口，将结果的行映射到 Java 对象（见示例 A-9）。

示例A-9 事务性JDBC服务中常见类型的Java配置

```
package basics;
```

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.SimpleDriverDataSource;
import org.springframework.jdbc.datasource.init.DataSourceInitializer;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;
import org.springframework.transaction.PlatformTransactionManager;
```

```
import javax.sql.DataSource;
```

```
@Configuration
```

```
public class TransactionalConfiguration {
```

①

```
@Bean
```

```
DataSource dataSource() {
    SimpleDriverDataSource dataSource = new SimpleDriverDataSource();
    dataSource
        .setUrl("jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE");
    dataSource.setDriverClass(org.h2.Driver.class);
    dataSource.setUsername("sa");
    dataSource.setPassword("");
    return dataSource;
}
```

②

```
@Bean
```

```
DataSourceInitializer dataSourceInitializer(DataSource ds,
@Value("classpath:/schema.sql") Resource schema,
@Value("classpath:/data.sql") Resource data) {
    DataSourceInitializer init = new DataSourceInitializer();
    init.setDatabasePopulator(new ResourceDatabasePopulator(schema, data));
    init.setDataSource(ds);
    return init;
}
```



```

③
@Bean
JdbcTemplate jdbcTemplate(dataSource ds) {
    return new JdbcTemplate(ds);
}

④
@Bean
RowMapper<Customer> customerRowMapper() {
    return (rs, i) -> new Customer(rs.getLong("ID"), rs.getString("FIRST_NAME"),
        rs.getString("LAST_NAME"));
}

⑤
@Bean
PlatformTransactionManager transactionManager(dataSource ds) {
    return new DataSourceTransactionManager(ds);
}
}

```

- ① 定义一个只与内存中的嵌入式 H2 数据库通信的 DataSource。
- ② 定义运行模式和数据初始化 DDL 的 DataSourceInitializer。
- ③ 一个 Spring JdbcTemplate，它将普通的 JDBC 调用减少到一行。
- ④ RowMapper 是一个回调接口，JdbcTemplate 用来将 SQL ResultSet 对象映射到对象中，在本示例中，类型为 Customer。
- ⑤ DataSourceTransactionManager bean 使 DataSource 事务适应 Spring 的 PlatformTransactionManager 层次结构。

我们可以使用 Spring 的低级 TransactionTemplate 来显式地划分事务边界。

基于TransactionTemplate的服务的Java配置

```

package basics.template;

import basics.Customer;
import basics.TransactionalConfiguration;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.*;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;

```

```
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;
```

```
@Configuration
@ComponentScan
@Import(TransactionalConfiguration.class)
public class TransactionTemplateApplication {

    public static void main(String args[]) {
        new AnnotationConfigApplicationContext(TransactionTemplateApplication.class);
    }
}
```

❶

```
@Bean
TransactionTemplate transactionTemplate(PlatformTransactionManager txManager) {
    return new TransactionTemplate(txManager);
}
}
```

```
@Service
class CustomerService {
```

```
    private JdbcTemplate jdbcTemplate;

    private TransactionTemplate txTemplate;

    private RowMapper<Customer> customerRowMapper;
```

```
@Autowired
public CustomerService(TransactionTemplate txTemplate,
    JdbcTemplate jdbcTemplate, RowMapper<Customer> customerRowMapper) {
    this.txTemplate = txTemplate;
    this.jdbcTemplate = jdbcTemplate;
    this.customerRowMapper = customerRowMapper;
}
}
```

```
public Customer enableCustomer(Long id) {
```

❷

```
TransactionCallback<Customer> customerTransactionCallback = (
    TransactionStatus transactionStatus) -> {
```

```
    String updateQuery = "update CUSTOMER set ENABLED = ? WHERE ID = ?";
    jdbcTemplate.update(updateQuery, Boolean.TRUE, id);
```

```
    String selectQuery = "select * from CUSTOMER where ID = ?";
```



```

        return jdbcTemplate.queryForObject(selectQuery, customerRowMapper, id);
    };

    ③
    Customer customer = txTemplate.execute(customerTransactionCallback);

    LogFactory.getLog(getClass())
        .info("retrieved customer # " + customer.getId());

    return customer;
}
}

```

- ① TransactionTemplate 只需要一个 PlatformTransactionManager。
- ② 使用 Java 8 lambdas 定义的 TransactionCallback。这个方法的主题将在一个有效的事务中运行，并在完成时被提交和关闭。
- ③ 返回值可以是任何你想要的值，但我认为返回值本身就是一个有趣的线索：事务最终应该是获得一个有效的副产品。

如果你曾经使用过 Spring 的其他 *Template 实现，应该对 TransactionTemplate 很熟悉。TransactionTemplate 明确定义了事务边界。当你想要封闭事务中的逻辑部分，并控制事务何时开始和停止时，这非常有用。

使用注解来声明式地划分事务边界。Spring 一直支持声明式事务规则，这两个规则都适用于业务组件，而且是内联的。在 Java SE 5 之后的 2005 年 5 月发布的定义事务规则最流行的方法是使用 Java 注解。

Spring 支持使用 @Transactional 注解 (<http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html>) 进行声明式事务管理。可以将它放置在一个类型或个别的方法前面。注解可用于指定事务性质，例如传播、回滚的异常等。

那么，使用 Java EE 做什么呢？在 2006 年发布的 Java EE 5 包含了 EJB 3，它定义了一种基于注解的方式来使用 javax.ejb.TransactionAttribute 注解标记事务的边界。如果 Spring bean 在 Spring bean 上被发现，Spring 也将履行这个注解。对于基于 EJB 的业务组件，TransactionAttribute 非常适用，但是这种方法在基于 EJB 的组件之外有点限制。JTA 1.2 定义了 javax.transaction.Transactional 作为一个通用的事务边界注解，其有点像 8 年前 Spring 的 @Transactional。如果有的话，Spring 也会履行这个注解。

除了使用 `@EnableTransactionManagement` 注解打开基于注解的事务边界，并且不再需要 `TransactionTemplate` bean 之外，使其工作的配置与 `TransactionTemplate` 基本相同。如果你使用的是 Spring Boot，那么你不需要添加 `@EnableTransactionManagement`；如果你配置了一个 `PlatformTransactionManager` bean（示例 A-10），它就会工作。

示例A-10 简单的基于@Transactional的应用程序的Java配置

```
package basics.annotation;

import basics.Customer;
import basics.TransactionalConfiguration;
import basics.template.TransactionTemplateApplication;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@ComponentScan
@Import({TransactionalConfiguration.class})
@EnableTransactionManagement
1 public class TransactionalApplication {

    public static void main(String args[]) {
        new AnnotationConfigApplicationContext(TransactionTemplateApplication.class);
    }
}

@Service
class CustomerService {

    private JdbcTemplate jdbcTemplate;

    private RowMapper<Customer> customerRowMapper;

    @Autowired
    public CustomerService(JdbcTemplate jdbcTemplate,
        RowMapper<Customer> customerRowMapper) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```



```

    this.customerRowMapper = customerRowMapper;
}

// @org.springframework.transaction.annotation.Transactional
❷
// @javax.ejb.TransactionAttribute ❸
@javax.transaction.Transactional
❹
public Customer enableCustomer(Long id) {

    String updateQuery = "update CUSTOMER set ENABLED = ? WHERE ID = ?";
    jdbcTemplate.update(updateQuery, Boolean.TRUE, id);

    String selectQuery = "select * from CUSTOMER where ID = ?";
    Customer customer = jdbcTemplate.queryForObject(selectQuery,
        customerRowMapper, id);

    LoggerFactory.getLog(getClass())
        .info("retrieved customer # " + customer.getId());
    return customer;
}
}

```

- ❶ @EnableTransactionManagement 打开事务处理。它要求在某个地方定义一个有效的 PlatformTransactionManager。
- ❷ 你可以使用 Spring 的 @org.springframework.transaction.annotation.Transactional。
- ❸ 或者 EJB 的 @javax.ejb.TransactionAttribute。
- ❹ 或者 JTA 1.2 的 @javax.transaction.Transactional。

我们倾向于使用 Spring 的 @Transactional 变种。它暴露比 EJB3 更多的能力，因为它能够公开 EJB 特定的 @TransactionAttribute（实际上 EJB 本身）不公开的事务性语义（比如事务挂起和恢复），并且不需要额外的 JTA 或 EJB 依赖。不过，无论如何，知道它会起作用都是很好的。

这些例子只需要使用一个 DataSource 驱动程序和 org.springframework.boot:spring-boot-starter-jdbc。

JTA 的全局事务

Spring 使得事务资源易于使用。但是，当试图以事务方式处理多个事务资源（例如全局

事务)时,开发人员要做些什么呢?例如,开发人员应如何事务性地数据库写入和确认接收到的 JMS 消息?全局事务是本地资源事物的替代品,它们涉及事务中的多个资源。为确保全局事务的完整性,协调员必须是独立于交易中资源的代理。它必须能够保证可以重放一个失败的事务,并且它本身是不会失败的。JTA (必然)增加了本地资源事务避免过程的复杂性和状态。大多数全局事务管理者都使用 X/Open XA (http://en.wikipedia.org/wiki/X/Open_XA) 协议,它允许事务资源(如数据库或消息代理)参与全局事务。用于 X/Open 协议的 Java EE 中间件称为 JTA (http://en.wikipedia.org/wiki/Java_Transaction_API)。

为此,只要知道 JTA 公开了两个关键接口,即 `javax.transaction.UserTransaction` (<http://docs.oracle.com/javaee/7/api/javax/transaction/UserTransaction.html>) 和 `javax.transaction.TransactionManager` (<http://docs.oracle.com/javaee/7/api/javax/transaction/TransactionManager.html>) 就可以了。

`UserTransaction` 支持常用的方法: `begin()`、`commit()`、`rollback()` 等。客户端使用这个对象来开始全局事务。当全局事务处于打开状态时, JTA 感知的资源(例如,支持 JTA 的 JDBC `javax.sql.XADataSource` 或支持 JTA 的 JMS `javax.jms.XAConnectionFactory`) 可以在 JTA 事务中注册。它们与全局事务的协调员进行通信,遵循一个协议,以原子的方式投入所有资源的工作或回滚。这取决于每个使用 XA 协议的原子资源是否支持 JTA 协议。

在 Java EE 容器中, `UserTransaction` 对象必须存在于 `java:comp/UserTransaction` 绑定下的 JNDI 上下文中,所以 Spring 的 `JtaTransactionManager` 很容易找到它。这个简单的接口足以处理基本的事务管理。值得注意的是,它不够复杂,无法处理子事务、事务挂起和恢复或者现代 JTA 实现中典型的其他问题。相反,如果可用,则使用 `TransactionManager` 实例。Java EE 应用程序服务器不需要公开此接口,并且很少在相同的 JNDI 下公开绑定。Spring 知道许多流行的 Java EE 应用程序服务器的众所周知的上下文,并会尝试自动为你找到它们。



通常,实现 `javax.transaction.UserTransaction` 的 bean 的同时也实现 `javax.transaction.TransactionManager`!

JTA 也可以在 Java EE 容器之外运行。有许多流行的第三方(和开源) JTA 实现,如 Atomikos 和 Bitronix。Spring Boot 为它们提供自动配置 (<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-jta>)。Atomikos (<http://www.atomikos.com>) 获得了商业支持,并提供开源版本。我们来看一个 `GreetingService` 中

的例子——它使用 JMS 发送通知，JPA 将记录作为全局事务的一部分持久化到 RDBMS（见示例 A-11）。

示例A-11 通过JTA同时使用JDBC和JMS资源的示例服务

```
package demo;

import org.springframework.jms.core.JmsTemplate;

import javax.inject.Inject;
import javax.inject.Named;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.Collection;

@Named
@Transactional
1 public class GreetingService {

    @Inject
    private JmsTemplate jmsTemplate; 2

    @PersistenceContext
    3 private EntityManager entityManager;

    public void createGreeting(String name, boolean fail) { 4
        Greeting greeting = new Greeting(name);
        this.entityManager.persist(greeting);
        this.jmsTemplate.convertAndSend("greetings", greeting);
        if (fail) {
            throw new RuntimeException("simulated error");
        }
    }

    public void createGreeting(String name) {
        this.createGreeting(name, false);
    }

    public Collection<Greeting> findAll() {
        return this.entityManager.createQuery(
            "select g from " + Greeting.class.getName() + " g", Greeting.class)
            .getResultList();
    }

    public Greeting find(Long id) {
```

```

    return this.entityManager.find(Greeting.class, id);
}
}

```

- ❶ 我们告诉 Spring，组件上的所有公共方法都是事务性的。
- ❷ 这段代码使用 Spring 的 JmsTemplate JMS 客户端来处理 JMS 资源。
- ❸ 这段代码使用 spring-boot-starter-data-jpa 支持，所以可以按照普通的 Java EE 惯例向 JPA 的 @PersistenceContext 注解注入一个 JPA EntityManager。
- ❹ 这个方法使用一个布尔值（如果是 true）触发一个异常。这个异常触发了 JTA 事务的回滚。你将只能看到在代码运行之后完成的三件工作中的两件。

我们通过在第三个事务中创建三个事务并模拟回滚来在 GreetingServiceClient 中证明这一点。你应该从控制台可以看到，有两条记录从 JDBC javax.sql.DataSource 数据源返回，并从嵌入式 JMS javax.jms.Destination 目标接收到两条记录（见示例 A-12）。

示例A-12 事务服务

```
package demo;
```

```

import javax.annotation.PostConstruct;
import javax.inject.Inject;
import javax.inject.Named;
import java.util.logging.Logger;

```

```

@Named
public class GreetingServiceClient {

    @Inject
    private GreetingService greetingService;

```

```

❶
    @PostConstruct
    public void afterPropertiesSet() throws Exception {
        greetingService.createGreeting("Phil");
        greetingService.createGreeting("Dave");
        try {
            greetingService.createGreeting("Josh", true);
        }
        catch (RuntimeException re) {
            Logger.getLogger(Application.class.getName()).info("caught exception...");
        }
        greetingService.findAll().forEach(System.out::println);
    }
}

```


- ① 我们还没见过 `@PostConstruct`。它是 JSR 250 的一部分，在语义上和 Spring 的 `InitializingBean` 接口相同。它定义了一个回调函数，在 bean 的依赖关系之后被调用，这些依赖关系为构造函数参数、JavaBean 特性或者字段。



Spring Boot 将根据配置的 `DataSource` 自动设置 JPA。这个例子使用 Spring Boot 的嵌入式 `DataSource` 支持 (<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-configure-a-datasource>)。如果嵌入式数据库（如 H2，<http://www.h2database.com/html/main.html>，这里使用它；或者 Derby 和 HSQL）在类路径中，并且没有显式定义 `javax.sql.DataSource`，则 Spring Boot 将创建一个 `DataSource` bean。

Spring Boot 使得建立一个 JMS 连接变得非常简单，就像嵌入的 `DataSource` 一样，Spring Boot 也可以创建一个嵌入式 JMS `ConnectionFactory` (<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#bootfeatures-hornetq>) 使用红帽的 HornetQ 消息代理来嵌入模式，假设正确的类型都已经在类路径（`org.springframework.boot:spring-boot-starter-hornetq`）下，并指定了一些属性（示例 A-13）。

示例A-13 配置嵌入式JMS ConnectionFactory

```
spring.hornetq.mode=embedded
spring.hornetq.embedded.enabled=true
spring.hornetq.embedded.queues=greetings
```

添加必要的 Spring Boot starter（`org.springframework.boot:spring-boot-starter-hornetq`）和 HornetQ 支持（`org.hornetq:hornetq-jms-server`）来激活正确的自动配置。如果要连接到传统的非嵌入式实例，则可以直接指定 `spring.datasource.url` 和 `spring.hornetq.host` 等属性，或者提供适当类型的 `@Bean` 定义。

这个例子使用 Spring Boot Atomikos 启动程序（`org.springframework.boot:spring-boot-starter-atomikos`）来配置 Atomikos 和支持 XA 的 JMS 和 JDBC 资源。

在今天的分布式世界里，考虑全局事务管理架构 (http://www.eaipatterns.com/ramblings/18_starbucks.html)。分布式事务控制一个服务以独立节奏处理事务的能力。分布式事务意味着当状态理想地处于单一的微服务中时，状态在多个服务中被维护。在理想情况下，服务应该共享状态，而不是在数据库级别，是在 API 级别，在这种情况下，整个讨论是没有意义的：REST API 无论如何也是不支持 X/Open 协议的！还有其他的状态同步模式，以消息传递为中心，促进横向可伸缩性和时间维度解耦。在消息传递和集成的讨论中，你会发现更多关于消息传递的信息。

在 Java EE 环境中部署

本附录中的示例使用了 Red Hat Wildfly 应用服务器特别棒的 Undertow 嵌入式 Servlet 引擎，而不是（Spring Boot 的默认）Apache Tomcat。如果你愿意，你也可以使用 Jetty。要使用其他方法，可以在 `org.springframework.boot:spring-boot-starter-tomcat` 中定义一个显式的依赖关系，然后将它的 `scope` 设置为 `provided`。这将会确保 Tomcat 不在类路径下，即使传递其中一个或多个其他依赖关系也是如此。然后，添加 `spring-boot-starter-jetty` 或 `spring-boot-starter-undertow` 的依赖项。尤其是 Undertow，其起源于第三方的 `pull request`，但作者真的很喜欢它，因为它启动起来非常快。

尽管我们的例子已经使用了很多我们相当熟悉的 Java EE API，但它仍然是典型的 Spring Boot，所以在默认情况下，你可以使用 `java -jar your.jar` 运行这个应用程序，或者将其部署到以流程为中心的 PaaS 产品（http://en.wikipedia.org/wiki/Platform_as_a_service）中，如 Heroku 或 Cloud Foundry（<http://cloudfoundry.org>）。

如果你想把它部署到一个独立的应用服务器上（比如 Apache Tomcat，或者 Websphere，或者其他的东西），把应用构建成一个 `.war`，并且相应地部署到任何 Servlet 3 容器中也很简单。在大多数情况下，就像将 Maven 构建的包更改为 `war` 一样简单，添加一个 Servlet 初始化器，并提供或者不包括提供 Servlet 容器的 Spring Boot 依赖（比如 `spring-boot-starter-tomcat` 或者 `spring-boot-starter-jetty`），该依赖 Java EE 容器将提供。如果使用 Spring Initializr，则可以从 Packaging 下拉列表中选择 `war` 跳过这些步骤。

如果你要转换现有的基于 `fat-jar` 的 Spring Boot 应用程序，则需要添加一个 Servlet 初始化程序类。这是 `web.xml` 的编程方式。Spring Boot 提供了一个基类 `org.springframework.boot.context.web.SpringBootServletInitializer`，它将以标准的 Servlet 3 方式支持 Spring Boot 和相关的机制。请注意，在部署应用程序时，任何嵌入式 Web 容器功能（如 SSL 配置、HTTP 资源压缩和端口管理）都将不起作用。

一个 Servlet 3 初始化类

```
package demo;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.support.SpringBootServletInitializer;

public class GreetingServletInitializer extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
```



```
    return builder.sources(Application.class);  
}  
}
```

如果将应用程序部署到更传统的应用程序服务器上，Spring Boot 就可以利用应用程序服务器的功能。使用 JNDI 绑定的 JMS ConnectionFactory、JDBC DataSource 或 JTA UserTransaction 是很简单的。

总结

我们会质疑这些 API。你真的需要分布式的多资源事务吗？JPA 是用于与基于 SQL 的 `javax.sql.DataSource` 通信的一个很好的 API，Spring Data 存储库当然包含了对 JPA 的支持，并且可以进一步发展。它们大大简化了 JPA。它们还支持 Cassandra、MongoDB、Redis、Riak、Couchbase、Neo4j，以及越来越多的其他技术。事实上，即使你的目标是使用带有 SQL 和 RDBMS 的 ORM，你仍然有很多好的选择。例如，MyBatis (<http://bit.ly/2vnpUmP>) 和 JOOQ (<https://www.jooq.org>) 的相关选项，这二者都可以作为 Spring Initializr (<http://start.spring.io>)。希望较低级别的 API 是通向微服务架构的途径而不是目的地。

在微服务环境中，Java EE 与现代微服务架构所需要的东西背道而驰。微服务是关于小而单独的服务，尽可能少的移动部分。一些现代的应用程序服务器可能非常小，但主体是一样的：为什么为你不需要的东西而付出？为什么到目前为止，只是为了满足运行代码的基础架构的不必要的耦合而让你的服务彼此分离？

应用程序服务器是在不同的时代（20 世纪 90 年代后期）建立起来的。应用程序服务器承诺隔离、监控以及整合的基础架构，但是现在很容易就可以看出，它没有做出特别好的工作。应用程序服务器不能保护应用程序不和类加载器之外的其他应用程序争用，甚至不是那么好，因此需要模块（Java 9 中的 Jigsaw 项目）和 OSGi 等技术。它不能隔离 CPU、RAM、文件系统，甚至 JVM 本身的组件也可能会饥饿。相反，让操作系统提供必要的隔离。操作系统关心和了解进程，而不是应用程序服务器和类加载器。

使用应用程序服务器配置的应用程序（甚至是应用程序服务器本身！）与应用程序一起检入源代码管理系统是非常常见的。这种（令人担忧的）方法意味着很少有人试图通过同一个应用程序服务器上合并应用程序来将 RAM 从机器中挤出来。他们这样做是因为他们想要构建和打包单个镜像，并且完成应用程序服务器的所有配置。为什么人工分离？如果没有机会将应用程序部署到任何其他应用程序服务器中，那为什么将两个工件（应用程序和服务）视为两个物理上分离的东西？

许多开发人员使用应用程序服务器（或像 Apache Tomcat 这样的 Web 服务器，主要是 Pivotal），因为它提供了一致的运维负担。运维知道如何部署、扩展和管理应用程序服务器。很公平。现在可以选择如 Docker 这样的容器技术或者是 Cloud Foundry 的 Warden 技术。容器为运维的操作提供了一致的管理平面。而容器里面运行的是什么则是完全不透明的。

容器和云，只关心进程。它不会假定你的应用程序会暴露一个 HTTP 端点。事实上也没有理由这样做。微服务没有任何可以表示 HTTP 和 REST 的东西，尽管它是通用的。通过从应用程序服务器转到基于 .jar 的部署，只要合适，就可以使用 HTTP（和一个 Servlet 容器）。我们喜欢这样想：如果 *du jour* 协议是 FTP 而不是 HTTP？如果全世界都通过 FTP 服务器访问暴露的服务呢？是不是感觉在架构上无法将我们的应用程序部署到 FTP 服务器呢？没有？那为什么我们将它们部署到 HTTP、EJB、JMS、JNDI 和 XA/Open 服务器？

在本附录中，我们看到，Spring 在 Java EE 服务和 API 方面表现良好，即使在微服务领域，使用这些 API 也可以得心应手。这里要告诉你的是，Java EE 服务器本身是不必要的，这只会放慢你的脚步。

关于作者

Josh Long 是一名 Spring 布道师，同时也是 InfoQ.com 的 Java queue 编辑，以及包括 *Spring Recipes* 第二版（Apress 出版社出版）在内的多本书籍的主要作者。Josh 在许多国际行业会议上发表过演讲，包括 TheServiceSide Java Symposium、SpringOne、OSCON、JavaZone、Devoxx、Java2Days 等。他不写 SpringSource 代码的时候，不是泡在 Java 用户组就是在咖啡店里喝咖啡。Josh 喜欢能够推动技术发展的解决方案。他的兴趣包括可扩展性、BPM、网格计算、移动计算和所谓的“智能”系统等。你可以在 blog.springsource.org 或 joshlong.com 上浏览他的博客。

Kenny Bastani 是 Pivotal 的 Spring 布道师。作为一名开源贡献者和博客作者，Kenny 较为关注图数据库、微服务等技术，并吸引了一群充满热情的软件开发人员。Kenny 还是 OSCON、SpringOne Platform 和 GOTO 等行业会议的常客。他维护了一个关于软件架构的个人博客（<http://kennybastani.com>），并提供构建事件驱动的微服务和无服务器架构的教程和开源参考示例。

封面介绍

Cloud Native Java 封面上的动物是蓝耳翠鸟（*Alcedo meninting*）。这种鸟被发现于亚洲，在印度次大陆和东南亚也有发现。

蓝耳翠鸟栖息于阴凉茂密的森林中，喜欢在小溪和水湾中狩猎。蓝耳翠鸟喜欢捕食栖息在浓密树枝上的甲壳类动物、蜻蜓幼虫等，也会潜水捕捉鱼类。其冠呈黑色，下半身颜色较深。其与其他翠鸟的主要区别是其耳朵上缺少条纹，居住地更加开放。成年雄性的喙颜色较暗；雌性的喙下半部分为红色。

O'Reilly 封面上的许多动物都是濒危物种，这些动物对世界很重要。想了解如何提供帮助，请浏览 animals.oreilly.com。

封面图片来自 *English Cyclopedia*。

云原生 Java

传统企业与亚马逊、Netflix和Etsy这类企业之间的区别是什么？这些公司有完善的云原生开发方法，这些方法使得他们能够保持优势并领先于竞争对手。本实践指南面向Java/JVM开发人员，详细讲解了如何使用Spring Boot、Spring Cloud和Cloud Foundry更快更好地构建软件。

很多组织都已踏足云计算、测试驱动开发、微服务与持续集成和持续交付领域。本书作者Josh Long和Kenny Bastani将带你深入研究这些领域中的工具和方法，并指导你将传统应用程序转变为真正的云原生应用程序。

本书包含以下四大部分。

- **基础知识：**了解云原生思维背后的动机；配置和测试Spring Boot应用程序；将传统应用程序迁移至云端。
- **微服务：**使用Spring构建HTTP和RESTful服务；在分布式系统中路由请求；建立更接近数据的边缘服务。
- **数据整合：**使用Spring Data管理数据，并将分布式服务与Spring支持的事件驱动的、以消息传递为中心的架构集成起来。
- **生产：**让你的系统可观测；使用服务代理来连接有状态的服务；了解持续交付背后的重要思想。

“本书是基于Java生态构建云原生应用的实战指南，几乎涵盖在构建云原生应用过程中可能遇到的所有问题，包括构建弹性服务、管理数据流（通过REST和异步事件）、测试、部署和可观测性等。”

——Daniel Bryant

SpectoLabs的软件开发人员
和CTO

“我相信，无论是你刚开启云原生之旅还是已经接近云原生的目标，都会从本书中获得一些经验和启发。”

——Dave Syer 博士

Spring框架的贡献者，
Spring Boot和Spring Cloud的
贡献者和联合创始人

Josh Long是Spring布道师，InfoQ.com的JavaQ编辑，是包括*Spring Recipes, 2e*（Apress出版社出版）在内的多本书籍的主要作者。你可以在blog.springsource.org和joshlong.com上阅读他的博客。

Kenny Bastani是Pivotal的Spring布道师，开源贡献者和软件架构博主（www.kennybastani.com）。Kenny为软件开发人员提供有关构建事件驱动的微服务和无服务器架构的教程。他还常在OSCON、SpringOne Platform和GOTO等行业会议上发表演讲。

图书分类：云计算

策划编辑：张春雨

责任编辑：牛勇



Broadview®
www.broadview.com.cn

O'Reilly Media, Inc. 授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

ISBN 978-7-121-34251-6



9 787121 342516 >

定价：128.00元